SciPy Reference Guide

Release 0.19.1

Written by the SciPy community

June 21, 2017

CONTENTS

Release 0.19.1
Date June 21, 2017

SciPy (pronounced "Sigh Pie") is open-source software for mathematics, science, and engineering.

CHAPTER

ONE

RELEASE NOTES

1.1 SciPy 0.19.1 Release Notes

SciPy 0.19.1 is a bug-fix release with no new features compared to 0.19.0. The most important change is a fix for a severe memory leak in `integrate.quad`.

1.1.1 Authors

- Evgeni Burovski
- Patrick Callier +
- Yu Feng
- Ralf Gommers
- Ilhan Polat
- Eric Quintero
- Scott Sievert
- Pauli Virtanen
- Warren Weckesser

A total of 9 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.19.1

- #7214: Memory use in integrate.quad in scipy-0.19.0
- #7258: `linalg.matrix_balance` gives wrong transformation matrix
- #7262: Segfault in daily testing
- #7273: `scipy.interpolate._bspl.evaluate_spline` gets wrong type
- #7335: scipy.signal.dlti(A,B,C,D).freqresp() fails

Pull requests for 0.19.1

- #7211: BUG: convolve may yield inconsistent dtypes with method changed
- #7216: BUG: integrate: fix refcounting bug in quad()
- #7229: MAINT: special: Rewrite a test of wrightomega
- #7261: FIX: Corrected the transformation matrix permutation
- #7265: BUG: Fix broken axis handling in spectral functions
- #7266: FIX 7262: ckdtree crashes in query_knn.
- #7279: Upcast half- and single-precision floats to doubles in BSpline...
- #7336: BUG: Fix signal.dfreqresp for StateSpace systems
- #7419: Fix several issues in `sparse.load_npz`, `save_npz`
- #7420: BUG: stats: allow integers as kappa4 shape parameters

1.2 SciPy 0.19.0 Release Notes

Contents

- *SciPy 0.19.0 Release Notes*
  - *New features*
    - *Foreign function interface improvements*
    - `scipy.linalg` *improvements*
    - `scipy.spatial` *improvements*
    - `scipy.ndimage` *improvements*
    - `scipy.optimize` *improvements*
    - `scipy.signal` *improvements*
    - `scipy.fftpack` *improvements*
    - `scipy.cluster` *improvements*
    - `scipy.sparse` *improvements*
    - `scipy.special` *improvements*
    - `scipy.stats` *improvements*
    - `scipy.interpolate` *improvements*
    - `scipy.integrate` *improvements*
  - *Deprecated features*
  - *Backwards incompatible changes*
  - *Other changes*
  - *Authors*

- *Issues closed for 0.19.0*
- *Pull requests for 0.19.0*

SciPy 0.19.0 is the culmination of 7 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.19.x branch, and on adding new features on the master branch.

This release requires Python 2.7 or 3.4-3.6 and NumPy 1.8.2 or greater.

Highlights of this release include:

- A unified foreign function interface layer, *scipy.LowLevelCallable*.
- Cython API for scalar, typed versions of the universal functions from the *scipy.special* module, via *cimport scipy.special.cython_special*.

1.2.1 New features

Foreign function interface improvements

scipy.LowLevelCallable provides a new unified interface for wrapping low-level compiled callback functions in the Python space. It supports Cython imported "api" functions, ctypes function pointers, CFFI function pointers, PyCapsules, Numba jitted functions and more. See gh-6509 for details.

scipy.linalg improvements

The function *scipy.linalg.solve* obtained two more keywords assume_a and transposed. The underlying LAPACK routines are replaced with "expert" versions and now can also be used to solve symmetric, hermitian and positive definite coefficient matrices. Moreover, ill-conditioned matrices now cause a warning to be emitted with the estimated condition number information. Old sym_pos keyword is kept for backwards compatibility reasons however it is identical to using assume_a='pos'. Moreover, the debug keyword, which had no function but only printing the overwrite_<a, b> values, is deprecated.

The function *scipy.linalg.matrix_balance* was added to perform the so-called matrix balancing using the LAPACK xGEBAL routine family. This can be used to approximately equate the row and column norms through diagonal similarity transformations.

The functions *scipy.linalg.solve_continuous_are* and *scipy.linalg.solve_discrete_are* have numerically more stable algorithms. These functions can also solve generalized algebraic matrix Riccati equations. Moreover, both gained a balanced keyword to turn balancing on and off.

scipy.spatial improvements

scipy.spatial.SphericalVoronoi.sort_vertices_of_regions has been re-written in Cython to improve performance.

scipy.spatial.SphericalVoronoi can handle > 200 k points (at least 10 million) and has improved performance.

The function *scipy.spatial.distance.directed_hausdorff* was added to calculate the directed Hausdorff distance.

`count_neighbors` method of *scipy.spatial.cKDTree* gained an ability to perform weighted pair counting via the new keywords `weights` and `cumulative`. See gh-5647 for details.

scipy.spatial.distance.pdist and *scipy.spatial.distance.cdist* now support non-double custom metrics.

`scipy.ndimage` improvements

The callback function C API supports PyCapsules in Python 2.7

Multidimensional filters now allow having different extrapolation modes for different axes.

`scipy.optimize` improvements

The *scipy.optimize.basinhopping* global minimizer obtained a new keyword, *seed*, which can be used to seed the random number generator and obtain repeatable minimizations.

The keyword *sigma* in *scipy.optimize.curve_fit* was overloaded to also accept the covariance matrix of errors in the data.

`scipy.signal` improvements

The function *scipy.signal.correlate* and *scipy.signal.convolve* have a new optional parameter *method*. The default value of *auto* estimates the fastest of two computation methods, the direct approach and the Fourier transform approach.

A new function has been added to choose the convolution/correlation method, *scipy.signal.choose_conv_method* which may be appropriate if convolutions or correlations are performed on many arrays of the same size.

New functions have been added to calculate complex short time fourier transforms of an input signal, and to invert the transform to recover the original signal: *scipy.signal.stft* and *scipy.signal.istft*. This implementation also fixes the previously incorrect ouput of *scipy.signal.spectrogram* when complex output data were requested.

The function *scipy.signal.sosfreqz* was added to compute the frequency response from second-order sections.

The function *scipy.signal.unit_impulse* was added to conveniently generate an impulse function.

The function *scipy.signal.iirnotch* was added to design second-order IIR notch filters that can be used to remove a frequency component from a signal. The dual function *scipy.signal.iirpeak* was added to compute the coefficients of a second-order IIR peak (resonant) filter.

The function *scipy.signal.minimum_phase* was added to convert linear-phase FIR filters to minimum phase.

The functions *scipy.signal.upfirdn* and *scipy.signal.resample_poly* are now substantially faster when operating on some n-dimensional arrays when n > 1. The largest reduction in computation time is realized in cases where the size of the array is small (<1k samples or so) along the axis to be filtered.

`scipy.fftpack` improvements

Fast Fourier transform routines now accept *np.float16* inputs and upcast them to *np.float32*. Previously, they would raise an error.

`scipy.cluster` improvements

Methods "centroid" and "median" of *scipy.cluster.hierarchy.linkage* have been significantly sped up. Long-standing issues with using `linkage` on large input data (over 16 GB) have been resolved.

`scipy.sparse` improvements

The functions *scipy.sparse.save_npz* and *scipy.sparse.load_npz* were added, providing simple serialization for some sparse formats.

The *prune* method of classes *bsr_matrix*, *csc_matrix*, and *csr_matrix* was updated to reallocate backing arrays under certain conditions, reducing memory usage.

The methods *argmin* and *argmax* were added to classes *coo_matrix*, *csc_matrix*, *csr_matrix*, and *bsr_matrix*.

New function *scipy.sparse.csgraph.structural_rank* computes the structural rank of a graph with a given sparsity pattern.

New function *scipy.sparse.linalg.spsolve_triangular* solves a sparse linear system with a triangular left hand side matrix.

`scipy.special` improvements

Scalar, typed versions of universal functions from *scipy.special* are available in the Cython space via `cimport` from the new module `scipy.special.cython_special`. These scalar functions can be expected to be significantly faster then the universal functions for scalar arguments. See the *scipy.special* tutorial for details.

Better control over special-function errors is offered by the functions *scipy.special.geterr* and *scipy.special.seterr* and the context manager *scipy.special.errstate*.

The names of orthogonal polynomial root functions have been changed to be consistent with other functions relating to orthogonal polynomials. For example, `scipy.special.j_roots` has been renamed *scipy.special.roots_jacobi* for consistency with the related functions *scipy.special.jacobi* and *scipy.special.eval_jacobi*. To preserve back-compatibility the old names have been left as aliases.

Wright Omega function is implemented as `scipy.special.wrightomega`.

`scipy.stats` improvements

The function *scipy.stats.weightedtau* was added. It provides a weighted version of Kendall's tau.

New class *scipy.stats.multinomial* implements the multinomial distribution.

New class *scipy.stats.rv_histogram* constructs a continuous univariate distribution with a piecewise linear CDF from a binned data sample.

New class *scipy.stats.argus* implements the Argus distribution.

`scipy.interpolate` improvements

New class *scipy.interpolate.BSpline* represents splines. `BSpline` objects contain knots and coefficients and can evaluate the spline. The format is consistent with FITPACK, so that one can do, for example:

```
>>> t, c, k = splrep(x, y, s=0)
>>> spl = BSpline(t, c, k)
>>> np.allclose(spl(x), y)
```

spl* functions, *scipy.interpolate.splev*, *scipy.interpolate.splint*, *scipy.interpolate.splder* and *scipy.interpolate.splantider*, accept both BSpline objects and (t, c, k) tuples for backwards compatibility.

For multidimensional splines, c.ndim > 1, BSpline objects are consistent with piecewise polynomials, *scipy.interpolate.PPoly*. This means that BSpline objects are not immediately consistent with *scipy.interpolate.splprep*, and one *cannot* do >>> BSpline(*splprep([x, y])[0]). Consult the *scipy.interpolate* test suite for examples of the precise equivalence.

In new code, prefer using scipy.interpolate.BSpline objects instead of manipulating (t, c, k) tuples directly.

New function *scipy.interpolate.make_interp_spline* constructs an interpolating spline given data points and boundary conditions.

New function *scipy.interpolate.make_lsq_spline* constructs a least-squares spline approximation given data points.

scipy.integrate improvements

Now *scipy.integrate.fixed_quad* supports vector-valued functions.

1.2.2 Deprecated features

scipy.interpolate.splmake, *scipy.interpolate.spleval* and *scipy.interpolate.spline* are deprecated. The format used by *splmake/spleval* was inconsistent with *splrep/splev* which was confusing to users.

scipy.special.errprint is deprecated. Improved functionality is available in *scipy.special.seterr*.

calling *scipy.spatial.distance.pdist* or *scipy.spatial.distance.cdist* with arguments not needed by the chosen metric is deprecated. Also, metrics *"old_cosine"* and *"old_cos"* are deprecated.

1.2.3 Backwards incompatible changes

The deprecated scipy.weave submodule was removed.

scipy.spatial.distance.squareform now returns arrays of the same dtype as the input, instead of always float64.

scipy.special.errprint now returns a boolean.

The function *scipy.signal.find_peaks_cwt* now returns an array instead of a list.

scipy.stats.kendalltau now computes the correct p-value in case the input contains ties. The p-value is also identical to that computed by *scipy.stats.mstats.kendalltau* and by R. If the input does not contain ties there is no change w.r.t. the previous implementation.

The function *scipy.linalg.block_diag* will not ignore zero-sized matrices anymore. Instead it will insert rows or columns of zeros of the appropriate size. See gh-4908 for more details.

1.2.4 Other changes

SciPy wheels will now report their dependency on numpy on all platforms. This change was made because Numpy wheels are available, and because the pip upgrade behavior is finally changing for the better (use --upgrade-strategy=only-if-needed for pip >= 8.2; that behavior will become the default in the next major version of pip).

Numerical values returned by `scipy.interpolate.interp1d` with `kind="cubic"` and `"quadratic"` may change relative to previous scipy versions. If your code depended on specific numeric values (i.e., on implementation details of the interpolators), you may want to double-check your results.

1.2.5 Authors

- @endolith
- Max Argus +
- Hervé Audren
- Alessandro Pietro Bardelli +
- Michael Benfield +
- Felix Berkenkamp
- Matthew Brett
- Per Brodtkorb
- Evgeni Burovski
- Pierre de Buyl
- CJ Carey
- Brandon Carter +
- Tim Cera
- Klesk Chonkin
- Christian Häggström +
- Luca Citi
- Peadar Coyle +
- Daniel da Silva +
- Greg Dooper +
- John Draper +
- drlvk +
- David Ellis +
- Yu Feng
- Baptiste Fontaine +
- Jed Frey +
- Siddhartha Gandhi +
- Wim Glenn +
- Akash Goel +
- Christoph Gohlke
- Ralf Gommers
- Alexander Goncearenco +
- Richard Gowers +

- Alex Griffing
- Radoslaw Guzinski +
- Charles Harris
- Callum Jacob Hays +
- Ian Henriksen
- Randy Heydon +
- Lindsey Hiltner +
- Gerrit Holl +
- Hiroki IKEDA +
- jfinkels +
- Mher Kazandjian +
- Thomas Keck +
- keuj6 +
- Kornel Kielczewski +
- Sergey B Kirpichev +
- Vasily Kokorev +
- Eric Larson
- Denis Laxalde
- Gregory R. Lee
- Josh Lefler +
- Julien Lhermitte +
- Evan Limanto +
- Jin-Guo Liu +
- Nikolay Mayorov
- Geordie McBain +
- Josue Melka +
- Matthieu Melot
- michaelvmartin15 +
- Surhud More +
- Brett M. Morris +
- Chris Mutel +
- Paul Nation
- Andrew Nelson
- David Nicholson +
- Aaron Nielsen +
- Joel Nothman

- nrnrk +
- Juan Nunez-Iglesias
- Mikhail Pak +
- Gavin Parnaby +
- Thomas Pingel +
- Ilhan Polat +
- Aman Pratik +
- Sebastian Pucilowski
- Ted Pudlik
- puenka +
- Eric Quintero
- Tyler Reddy
- Joscha Reimer
- Antonio Horta Ribeiro +
- Edward Richards +
- Roman Ring +
- Rafael Rossi +
- Colm Ryan +
- Sami Salonen +
- Alvaro Sanchez-Gonzalez +
- Johannes Schmitz
- Kari Schoonbee
- Yurii Shevchuk +
- Jonathan Siebert +
- Jonathan Tammo Siebert +
- Scott Sievert +
- Sourav Singh
- Byron Smith +
- Srikiran +
- Samuel St-Jean +
- Yoni Teitelbaum +
- Bhavika Tekwani
- Martin Thoma
- timbalam +
- Svend Vanderveken +
- Sebastiano Vigna +

- Aditya Vijaykumar +
- Santi Villalba +
- Ze Vinicius
- Pauli Virtanen
- Matteo Visconti
- Yusuke Watanabe +
- Warren Weckesser
- Phillip Weinberg +
- Nils Werner
- Jakub Wilk
- Josh Wilson
- wirew0rm +
- David Wolever +
- Nathan Woods
- ybeltukov +
- G Young
- Evgeny Zhurko +

A total of 121 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.19.0

- #1767: Function definitions in __fitpack.h should be moved. (Trac #1240)
- #1774: _kmeans chokes on large thresholds (Trac #1247)
- #2089: Integer overflows cause segfault in linkage function with large...
- #2190: Are odd-length window functions supposed to be always symmetrical?...
- #2251: solve_discrete_are in scipy.linalg does (sometimes) not solve...
- #2580: scipy.interpolate.UnivariateSpline (or a new superclass of it)...
- #2592: scipy.stats.anderson assumes gumbel_l
- #3054: scipy.linalg.eig does not handle infinite eigenvalues
- #3160: multinomial pmf / logpmf
- #3904: scipy.special.ellipj dn wrong values at quarter period
- #4044: Inconsistent code book initialization in kmeans
- #4234: scipy.signal.flattop documentation doesn't list a source for...
- #4831: Bugs in C code in __quadpack.h
- #4908: bug: unnessesary validity check for block dimension in scipy.sparse.block_diag
- #4917: BUG: indexing error for sparse matrix with **ix_**

- #4938: Docs on extending ndimage need to be updated.
- #5056: sparse matrix element-wise multiplying dense matrix returns dense...
- #5337: Formula in documentation for correlate is wrong
- #5537: use OrderedDict in io.netcdf
- #5750: [doc] missing data index value in KDTree, cKDTree
- #5755: p-value computation in scipy.stats.kendalltau() in broken in...
- #5757: BUG: Incorrect complex output of signal.spectrogram
- #5964: ENH: expose scalar versions of scipy.special functions to cython
- #6107: scipy.cluster.hierarchy.single segmentation fault with 2**16...
- #6278: optimize.basinhopping should take a RandomState object
- #6296: InterpolatedUnivariateSpline: check_finite fails when w is unspecified
- #6306: Anderson-Darling bad results
- #6314: scipy.stats.kendaltau() p value not in agreement with R, SPSS...
- #6340: Curve_fit bounds and maxfev
- #6377: expm_multiply, complex matrices not working using start,stop,ect...
- #6382: optimize.differential_evolution stopping criterion has unintuitive...
- #6391: Global Benchmarking times out at 600s.
- #6397: mmwrite errors with large (but still 64-bit) integers
- #6413: scipy.stats.dirichlet computes multivariate gaussian differential...
- #6428: scipy.stats.mstats.mode modifies input
- #6440: Figure out ABI break policy for scipy.special Cython API
- #6441: Using Qhull for halfspace intersection : segfault
- #6442: scipy.spatial : In incremental mode volume is not recomputed
- #6451: Documentation for scipy.cluster.hierarchy.to_tree is confusing...
- #6490: interp1d (kind=zero) returns wrong value for rightmost interpolation...
- #6521: scipy.stats.entropy does *not* calculate the KL divergence
- #6530: scipy.stats.spearmanr unexpected NaN handling
- #6541: Test runner does not run scipy._lib/tests?
- #6552: BUG: misc.bytescale returns unexpected results when using cmin/cmax...
- #6556: RectSphereBivariateSpline(u, v, r) fails if min(v) >= pi
- #6559: Differential_evolution maxiter causing memory overflow
- #6565: Coverage of spectral functions could be improved
- #6628: Incorrect parameter name in binomial documentation
- #6634: Expose LAPACK's xGESVX family for linalg.solve ill-conditioned...
- #6657: Confusing documentation for `scipy.special.sph_harm`
- #6676: optimize: Incorrect size of Jacobian returned by 'minimize(...,...

- #6681: add a new context manager to wrap `scipy.special.seterr`
- #6700: BUG: scipy.io.wavfile.read stays in infinite loop, warns on wav...
- #6721: scipy.special.chebyt(N) throw a 'TypeError' when N > 64
- #6727: Documentation for scipy.stats.norm.fit is incorrect
- #6764: Documentation for scipy.spatial.Delaunay is partially incorrect
- #6811: scipy.spatial.SphericalVoronoi fails for large number of points
- #6841: spearmanr fails when nan_policy='omit' is set
- #6869: Currently in gaussian_kde, the logpdf function is calculated...
- #6875: SLSQP inconsistent handling of invalid bounds
- #6876: Python stopped working (Segfault?) with minimum/maximum filter...
- #6889: dblquad gives different results under scipy 0.17.1 and 0.18.1
- #6898: BUG: dblquad ignores error tolerances
- #6901: Solving sparse linear systems in CSR format with complex values
- #6903: issue in spatial.distance.pdist docstring
- #6917: Problem in passing drop_rule to scipy.sparse.linalg.spilu
- #6926: signature mismatches for LowLevelCallable
- #6961: Scipy contains shebang pointing to /usr/bin/python and /bin/bash...
- #6972: BUG: special: *generate_ufuncs.py* is broken
- #6984: Assert raises test failure for test_ill_condition_warning
- #6990: BUG: sparse: Bad documentation of the *k* argument in *sparse.linalg.eigs*
- #6991: Division by zero in linregress()
- #7011: possible speed improvment in rv_continuous.fit()
- #7015: Test failure with Python 3.5 and numpy master
- #7055: SciPy 0.19.0rc1 test errors and failures on Windows
- #7096: macOS test failues for test_solve_continuous_are
- #7100: test_distance.test_Xdist_deprecated_args test error in 0.19.0rc2

Pull requests for 0.19.0

- #2908: Scipy 1.0 Roadmap
- #3174: add b-splines
- #4606: ENH: Add a unit impulse waveform function
- #5608: Adds keyword argument to choose faster convolution method
- #5647: ENH: Faster count_neighour in cKDTree / + weighted input data
- #6021: Netcdf append
- #6058: ENH: scipy.signal - Add stft and istft
- #6059: ENH: More accurate signal.freqresp for zpk systems

- #6195: ENH: Cython interface for special
- #6234: DOC: Fixed a typo in ward() help
- #6261: ENH: add docstring and clean up code for signal.normalize
- #6270: MAINT: special: add tests for cdflib
- #6271: Fix for scipy.cluster.hierarchy.is_isomorphic
- #6273: optimize: rewrite while loops as for loops
- #6279: MAINT: Bessel tweaks
- #6291: Fixes gh-6219: remove runtime warning from genextreme distribution
- #6294: STY: Some PEP8 and cleaning up imports in stats/_continuous_distns.py
- #6297: Clarify docs in misc/__init__.py
- #6300: ENH: sparse: Loosen input validation for *diags* with empty inputs
- #6301: BUG: standardizes check_finite behavior re optional weights,...
- #6303: Fixing example in _lazyselect docstring.
- #6307: MAINT: more improvements to gammainc/gammaincc
- #6308: Clarified documentation of hypergeometric distribution.
- #6309: BUG: stats: Improve calculation of the Anderson-Darling statistic.
- #6315: ENH: Descending order of x in PPoly
- #6317: ENH: stats: Add support for nan_policy to stats.median_test
- #6321: TST: fix a typo in test name
- #6328: ENH: sosfreqz
- #6335: Define LinregressResult outside of linregress
- #6337: In anderson test, added support for right skewed gumbel distribution.
- #6341: Accept several spellings for the curve_fit max number of function...
- #6342: DOC: cluster: clarify hierarchy.linkage usage
- #6352: DOC: removed brentq from its own 'see also'
- #6362: ENH: stats: Use explicit formulas for sf, logsf, etc in weibull...
- #6369: MAINT: special: add a comment to hyp0f1_complex
- #6375: Added the multinomial distribution.
- #6387: MAINT: special: improve accuracy of ellipj's *dn* at quarter...
- #6388: BenchmarkGlobal - getting it to work in Python3
- #6394: ENH: scipy.sparse: add save and load functions for sparse matrices
- #6400: MAINT: moves global benchmark run from setup_cache to track_all
- #6403: ENH: seed kwd for basinhopping. Closes #6278
- #6404: ENH: signal: added irrnotch and iirpeak functions.
- #6406: ENH: special: extend *sici*/*shichi* to complex arguments
- #6407: ENH: Window functions should not accept non-integer or negative...

- #6408: MAINT: _differentialevolution now uses _lib._util.check_random_state
- #6427: MAINT: Fix gmpy build & test that mpmath uses gmpy
- #6439: MAINT: ndimage: update callback function c api
- #6443: BUG: Fix volume computation in incremental mode
- #6447: Fixes issue #6413 - Minor documentation fix in the entropy function...
- #6448: ENH: Add halfspace mode to Qhull
- #6449: ENH: rtol and atol for differential_evolution termination fixes...
- #6453: DOC: Add some See Also links between similar functions
- #6454: DOC: linalg: clarify callable signature in *ordqz*
- #6457: ENH: spatial: enable non-double dtypes in squareform
- #6459: BUG: Complex matrices not handled correctly by expm_multiply...
- #6465: TST DOC Window docs, tests, etc.
- #6469: ENH: linalg: better handling of infinite eigenvalues in *eig*/*eigvals*
- #6475: DOC: calling interp1d/interp2d with NaNs is undefined
- #6477: Document magic numbers in optimize.py
- #6481: TST: Supress some warnings from test_windows
- #6485: DOC: spatial: correct typo in procrustes
- #6487: Fix Bray-Curtis formula in pdist docstring
- #6493: ENH: Add covariance functionality to scipy.optimize.curve_fit
- #6494: ENH: stats: Use log1p() to improve some calculations.
- #6495: BUG: Use MST algorithm instead of SLINK for single linkage clustering
- #6497: MRG: Add minimum_phase filter function
- #6505: reset scipy.signal.resample window shape to 1-D
- #6507: BUG: linkage: Raise exception if y contains non-finite elements
- #6509: ENH: _lib: add common machinery for low-level callback functions
- #6520: scipy.sparse.base.__mul__ non-numpy/scipy objects with 'shape'...
- #6522: Replace kl_div by rel_entr in entropy
- #6524: DOC: add next_fast_len to list of functions
- #6527: DOC: Release notes to reflect the new covariance feature in optimize.curve_fit
- #6532: ENH: Simplify _cos_win, document it, add symmetric/periodic arg
- #6535: MAINT: sparse.csgraph: updating old cython loops
- #6540: DOC: add to documentation of orthogonal polynomials
- #6544: TST: Ensure tests for scipy._lib are run by scipy.test()
- #6546: updated docstring of stats.linregress
- #6553: commited changes that I originally submitted for scipy.signal.cspline…
- #6561: BUG: modify signal.find_peaks_cwt() to return array and accept...

- #6562: DOC: Negative binomial distribution clarification
- #6563: MAINT: be more liberal in requiring numpy
- #6567: MAINT: use xrange for iteration in differential_evolution fixes...
- #6572: BUG: "sp.linalg.solve_discrete_are" fails for random data
- #6578: BUG: misc: allow both cmin/cmax and low/high params in bytescale
- #6581: Fix some unfortunate typos
- #6582: MAINT: linalg: make handling of infinite eigenvalues in *ordqz*...
- #6585: DOC: interpolate: correct seealso links to ndimage
- #6588: Update docstring of scipy.spatial.distance_matrix
- #6592: DOC: Replace 'first' by 'smallest' in mode
- #6593: MAINT: remove scipy.weave submodule
- #6594: DOC: distance.squareform: fix html docs, add note about dtype...
- #6598: [DOC] Fix incorrect error message in medfilt2d
- #6599: MAINT: linalg: turn a *solve_discrete_are* test back on
- #6600: DOC: Add SOS goals to roadmap
- #6601: DEP: Raise minimum numpy version to 1.8.2
- #6605: MAINT: 'new' module is deprecated, don't use it
- #6607: DOC: add note on change in wheel dependency on numpy and pip...
- #6609: Fixes #6602 - Typo in docs
- #6616: ENH: generalization of continuous and discrete Riccati solvers...
- #6621: DOC: improve cluster.hierarchy docstrings.
- #6623: CS matrix prune method should copy data from large unpruned arrays
- #6625: DOC: special: complete documentation of *eval_** functions
- #6626: TST: special: silence some deprecation warnings
- #6631: fix parameter name doc for discrete distributions
- #6632: MAINT: stats: change some instances of *special* to *sc*
- #6633: MAINT: refguide: py2k long integers are equal to py3k integers
- #6638: MAINT: change type declaration in cluster.linkage, prevent overflow
- #6640: BUG: fix issue with duplicate values used in cluster.vq.kmeans
- #6641: BUG: fix corner case in cluster.vq.kmeans for large thresholds
- #6643: MAINT: clean up truncation modes of dendrogram
- #6645: MAINT: special: rename **_roots* functions
- #6646: MAINT: clean up mpmath imports
- #6647: DOC: add sqrt to Mahalanobis description for pdist
- #6648: DOC: special: add a section on *cython_special* to the tutorial
- #6649: ENH: Added scipy.spatial.distance.directed_hausdorff

- #6650: DOC: add Sphinx roles for DOI and arXiv links
- #6651: BUG: mstats: make sure mode(..., None) does not modify its input
- #6652: DOC: special: add section to tutorial on functions not in special
- #6653: ENH: special: add the Wright Omega function
- #6656: ENH: don't coerce input to double with custom metric in cdist...
- #6658: Faster/shorter code for computation of discordances
- #6659: DOC: special: make __init__ summaries and html summaries match
- #6661: general.rst: Fix a typo
- #6664: TST: Spectral functions' window correction factor
- #6665: [DOC] Conditions on v in RectSphereBivariateSpline
- #6668: DOC: Mention negative masses for center of mass
- #6675: MAINT: special: remove outdated README
- #6677: BUG: Fixes computation of p-values.
- #6679: BUG: optimize: return correct Jacobian for method 'SLSQP' in...
- #6680: ENH: Add structural rank to sparse.csgraph
- #6686: TST: Added Airspeed Velocity benchmarks for SphericalVoronoi
- #6687: DOC: add section "deciding on new features" to developer guide.
- #6691: ENH: Clearer error when fmin_slsqp obj doesn't return scalar
- #6702: TST: Added airspeed velocity benchmarks for scipy.spatial.distance.cdist
- #6707: TST: interpolate: test fitpack wrappers, not _impl
- #6709: TST: fix a number of test failures on 32-bit systems
- #6711: MAINT: move function definitions from __fitpack.h to _fitpackmodule.c
- #6712: MAINT: clean up wishlist in stats.morestats, and copyright statement.
- #6715: DOC: update the release notes with BSpline et al.
- #6716: MAINT: scipy.io.wavfile: No infinite loop when trying to read...
- #6717: some style cleanup
- #6723: BUG: special: cast to float before in-place multiplication in...
- #6726: address performance regressions in interp1d
- #6728: DOC: made code examples in *integrate* tutorial copy-pasteable
- #6731: DOC: scipy.optimize: Added an example for wrapping complex-valued...
- #6732: MAINT: cython_special: remove *errprint*
- #6733: MAINT: special: fix some pyflakes warnings
- #6734: DOC: sparse.linalg: fixed matrix description in *bicgstab* doc
- #6737: BLD: update *cythonize.py* to detect changes in pxi files
- #6740: DOC: special: some small fixes to docstrings
- #6741: MAINT: remove dead code in interpolate.py

- #6742: BUG: fix `linalg.block_diag` to support zero-sized matrices.
- #6744: ENH: interpolate: make PPoly.from_spline accept BSpline objects
- #6746: DOC: special: clarify use of Condon-Shortley phase in *sph_harm*/*lpmv*
- #6750: ENH: sparse: avoid densification on broadcasted elem-wise mult
- #6751: sinm doc explained cosm
- #6753: ENH: special: allow for more fine-tuned error handling
- #6759: Move logsumexp and pade from scipy.misc to scipy.special and...
- #6761: ENH: argmax and argmin methods for sparse matrices
- #6762: DOC: Improve docstrings of sparse matrices
- #6763: ENH: Weighted tau
- #6768: ENH: cythonized spherical Voronoi region polygon vertex sorting
- #6770: Correction of Delaunay class' documentation
- #6775: ENH: Integrating LAPACK "expert" routines with conditioning warnings...
- #6776: MAINT: Removing the trivial f2py warnings
- #6777: DOC: Update rv_continuous.fit doc.
- #6778: MAINT: cluster.hierarchy: Improved wording of error msgs
- #6786: BLD: increase minimum Cython version to 0.23.4
- #6787: DOC: expand on `linalg.block_diag` changes in 0.19.0 release...
- #6789: ENH: Add further documentation for norm.fit
- #6790: MAINT: Fix a potential problem in nn_chain linkage algorithm
- #6791: DOC: Add examples to scipy.ndimage.fourier
- #6792: DOC: fix some numpydoc / Sphinx issues.
- #6793: MAINT: fix circular import after moving functions out of misc
- #6796: TST: test importing each submodule. Regression test for gh-6793.
- #6799: ENH: stats: Argus distribution
- #6801: ENH: stats: Histogram distribution
- #6803: TST: make sure tests for `_build_utils` are run.
- #6804: MAINT: more fixes in *loggamma*
- #6806: ENH: Faster linkage for 'centroid' and 'median' methods
- #6810: ENH: speed up upfirdn and resample_poly for n-dimensional arrays
- #6812: TST: Added ConvexHull asv benchmark code
- #6814: ENH: Different extrapolation modes for different dimensions in...
- #6826: Signal spectral window default fix
- #6828: BUG: SphericalVoronoi Space Complexity (Fixes #6811)
- #6830: RealData docstring correction
- #6834: DOC: Added reference for skewtest function. See #6829

- #6836: DOC: Added mode='mirror' in the docstring for the functions accepting...
- #6838: MAINT: sparse: start removing old BSR methods
- #6844: handle incompatible dimensions when input is not an ndarray in...
- #6847: Added maxiter to golden search.
- #6850: BUG: added check for optional param scipy.stats.spearmanr
- #6858: MAINT: Removing redundant tests
- #6861: DEP: Fix escape sequences deprecated in Python 3.6.
- #6862: DOC: dx should be float, not int
- #6863: updated documentation curve_fit
- #6866: DOC : added some documentation to j1 referring to spherical_jn
- #6867: DOC: cdist move long examples list into Notes section
- #6868: BUG: Make stats.mode return a ModeResult namedtuple on empty...
- #6871: Corrected documentation.
- #6874: ENH: gaussian_kde.logpdf based on logsumexp
- #6877: BUG: ndimage: guard against footprints of all zeros
- #6881: python 3.6
- #6885: Vectorized integrate.fixed_quad
- #6886: fixed typo
- #6891: TST: fix failures for linalg.dare/care due to tightened test...
- #6892: DOC: fix a bunch of Sphinx errors.
- #6894: TST: Added asv benchmarks for scipy.spatial.Voronoi
- #6908: BUG: Fix return dtype for complex input in spsolve
- #6909: ENH: fftpack: use float32 routines for float16 inputs.
- #6911: added min/max support to binned_statistic
- #6913: Fix 6875: SLSQP raise ValueError for all invalid bounds.
- #6914: DOCS: GH6903 updating docs of Spatial.distance.pdist
- #6916: MAINT: fix some issues for 32-bit Python
- #6924: BLD: update Bento build for scipy.LowLevelCallable
- #6932: ENH: Use OrderedDict in io.netcdf. Closes gh-5537
- #6933: BUG: fix LowLevelCallable issue on 32-bit Python.
- #6936: BUG: sparse: handle size-1 2D indexes correctly
- #6938: TST: fix test failures in special on 32-bit Python.
- #6939: Added attributes list to cKDTree docstring
- #6940: improve efficiency of dok_matrix.tocoo
- #6942: DOC: add link to liac-arff package in the io.arff docstring.
- #6943: MAINT: Docstring fixes and an additional test for linalg.solve

- #6944: DOC: Add example of odeint with a banded Jacobian to the integrate...
- #6946: ENH: hypergeom.logpmf in terms of betaln
- #6947: TST: speedup distance tests
- #6948: DEP: Deprecate the keyword "debug" from linalg.solve
- #6950: BUG: Correctly treat large integers in MMIO (fixes #6397)
- #6952: ENH: Minor user-friendliness cleanup in LowLevelCallable
- #6956: DOC: improve description of 'output' keyword for convolve
- #6957: ENH more informative error in sparse.bmat
- #6962: Shebang fixes
- #6964: DOC: note argmin/argmax addition
- #6965: BUG: Fix issues passing error tolerances in dblquad and tplquad.
- #6971: fix the docstring of signaltools.correlate
- #6973: Silence expected numpy warnings in scipy.ndimage.interpolation.zoom()
- #6975: BUG: special: fix regex in *generate_ufuncs.py*
- #6976: Update docstring for griddata
- #6978: Avoid division by zero in zoom factor calculation
- #6979: BUG: ARE solvers did not check the generalized case carefully
- #6985: ENH: sparse: add scipy.sparse.linalg.spsolve_triangular
- #6994: MAINT: spatial: updates to plotting utils
- #6995: DOC: Bad documentation of k in sparse.linalg.eigs See #6990
- #6997: TST: Changed the test with a less singular example
- #7000: DOC: clarify interp1d 'zero' argument
- #7007: BUG: Fix division by zero in linregress() for 2 data points
- #7009: BUG: Fix problem in passing drop_rule to scipy.sparse.linalg.spilu
- #7012: speed improvment in _distn_infrastructure.py
- #7014: Fix Typo: add a single quotation mark to fix a slight typo
- #7021: MAINT: stats: use machine constants from np.finfo, not machar
- #7026: MAINT: update .mailmap
- #7032: Fix layout of rv_histogram docs
- #7035: DOC: update 0.19.0 release notes
- #7036: ENH: Add more boundary options to signal.stft
- #7040: TST: stats: skip too slow tests
- #7042: MAINT: sparse: speed up setdiag tests
- #7043: MAINT: refactory and code cleaning Xdist
- #7053: Fix msvc 9 and 10 compile errors
- #7060: DOC: updated release notes with #7043 and #6656

- #7062: MAINT: Change defaut STFT boundary kwarg to "zeros"
- #7064: Fix ValueError: path is on mount 'X:', start on mount 'D:' on...
- #7067: TST: Fix PermissionError: [Errno 13] Permission denied on Windows
- #7068: TST: Fix UnboundLocalError: local variable 'data' referenced...
- #7069: Fix OverflowError: Python int too large to convert to C long...
- #7071: TST: silence RuntimeWarning for nan test of stats.spearmanr
- #7072: Fix OverflowError: Python int too large to convert to C long...
- #7084: TST: linalg: bump tolerance in test_falker
- #7095: TST: linalg: bump more tolerances in test_falker
- #7101: TST: Relax solve_continuous_are test case 2 and 12
- #7106: BUG: stop cdist "correlation" modifying input
- #7116: Backports to 0.19.0rc2

1.3 SciPy 0.18.1 Release Notes

SciPy 0.18.1 is a bug-fix release with no new features compared to 0.18.0.

1.3.1 Authors

- @kleskjr
- Evgeni Burovski
- CJ Carey
- Luca Citi +
- Yu Feng
- Ralf Gommers
- Johannes Schmitz +
- Josh Wilson
- Nathan Woods

A total of 9 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.18.1

- #6357: scipy 0.17.1 piecewise cubic hermite interpolation does not return...
- #6420: circmean() changed behaviour from 0.17 to 0.18
- #6421: scipy.linalg.solve_banded overwrites input 'b' when the inversion...
- #6425: cKDTree INF bug
- #6435: scipy.stats.ks_2samp returns different values on different computers

- #6458: Error in scipy.integrate.dblquad when using variable integration...

Pull requests for 0.18.1

- #6405: BUG: sparse: fix elementwise divide for CSR/CSC
- #6431: BUG: result for insufficient neighbours from cKDTree is wrong.
- #6432: BUG Issue #6421: scipy.linalg.solve_banded overwrites input 'b'...
- #6455: DOC: add links to release notes
- #6462: BUG: interpolate: fix .roots method of PchipInterpolator
- #6492: BUG: Fix regression in dblquad: #6458
- #6543: fix the regression in circmean
- #6545: Revert gh-5938, restore ks_2samp
- #6557: Backports for 0.18.1

1.4 SciPy 0.18.0 Release Notes

Contents

- *SciPy 0.18.0 Release Notes*
  - *New features*
    - `scipy.integrate` *improvements*
    - `scipy.interpolate` *improvements*
    - `scipy.fftpack` *improvements*
    - `scipy.signal` *improvements*
      - *Discrete-time linear systems*
    - `scipy.sparse` *improvements*
    - `scipy.optimize` *improvements*
    - `scipy.stats` *improvements*
      - *Random matrices*
    - `scipy.linalg` *improvements*
    - `scipy.spatial` *improvements*
    - `scipy.cluster` *improvements*
    - `scipy.special` *improvements*
  - *Deprecated features*
  - *Backwards incompatible changes*
    - `scipy.optimize`
    - `scipy.ndimage`

    * scipy.stats
    * scipy.io
    * scipy.interpolate
  - *Other changes*
  - *Authors*
    * *Issues closed for 0.18.0*
    * *Pull requests for 0.18.0*

SciPy 0.18.0 is the culmination of 6 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.19.x branch, and on adding new features on the master branch.

This release requires Python 2.7 or 3.4-3.5 and NumPy 1.7.1 or greater.

Highlights of this release include:

- A new ODE solver for two-point boundary value problems, *scipy.optimize.solve_bvp*.
- A new class, *CubicSpline*, for cubic spline interpolation of data.
- N-dimensional tensor product polynomials, `scipy.interpolate.NdPPoly`.
- Spherical Voronoi diagrams, `scipy.spatial.SphericalVoronoi`.
- Support for discrete-time linear systems, `scipy.signal.dlti`.

1.4.1 New features

`scipy.integrate` improvements

A solver of two-point boundary value problems for ODE systems has been implemented in `scipy.integrate.solve_bvp`. The solver allows for non-separated boundary conditions, unknown parameters and certain singular terms. It finds a C1 continious solution using a fourth-order collocation algorithm.

`scipy.interpolate` improvements

Cubic spline interpolation is now available via `scipy.interpolate.CubicSpline`. This class represents a piecewise cubic polynomial passing through given points and C2 continuous. It is represented in the standard polynomial basis on each segment.

A representation of n-dimensional tensor product piecewise polynomials is available as the `scipy.interpolate.NdPPoly` class.

Univariate piecewise polynomial classes, *PPoly* and *Bpoly*, can now be evaluated on periodic domains. Use `extrapolate="periodic"` keyword argument for this.

`scipy.fftpack` improvements

`scipy.fftpack.next_fast_len` function computes the next "regular" number for FFTPACK. Padding the input to this length can give significant performance increase for `scipy.fftpack.fft`.

`scipy.signal` improvements

Resampling using polyphase filtering has been implemented in the function `scipy.signal.resample_poly`. This method upsamples a signal, applies a zero-phase low-pass FIR filter, and downsamples using `scipy.signal.upfirdn` (which is also new in 0.18.0). This method can be faster than FFT-based filtering provided by `scipy.signal.resample` for some signals.

`scipy.signal.firls`, which constructs FIR filters using least-squares error minimization, was added.

`scipy.signal.sosfiltfilt`, which does forward-backward filtering like `scipy.signal.filtfilt` but for second-order sections, was added.

Discrete-time linear systems

`scipy.signal.dlti` provides an implementation of discrete-time linear systems. Accordingly, the *StateSpace*, *TransferFunction* and *ZerosPolesGain* classes have learned a the new keyword, *dt*, which can be used to create discrete-time instances of the corresponding system representation.

`scipy.sparse` improvements

The functions *sum*, *max*, *mean*, *min*, *transpose*, and *reshape* in `scipy.sparse` have had their signatures augmented with additional arguments and functionality so as to improve compatibility with analogously defined functions in `numpy`.

Sparse matrices now have a *count_nonzero* method, which counts the number of nonzero elements in the matrix. Unlike *getnnz()* and `nnz` propety, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

`scipy.optimize` improvements

The implementation of Nelder-Mead minimization, *scipy.minimize(..., method="Nelder-Mead")*, obtained a new keyword, *initial_simplex*, which can be used to specify the initial simplex for the optimization process.

Initial step size selection in CG and BFGS minimizers has been improved. We expect that this change will improve numeric stability of optimization in some cases. See pull request gh-5536 for details.

Handling of infinite bounds in SLSQP optimization has been improved. We expect that this change will improve numeric stability of optimization in the some cases. See pull request gh-6024 for details.

A large suite of global optimization benchmarks has been added to `scipy/benchmarks/go_benchmark_functions`. See pull request gh-4191 for details.

Nelder-Mead and Powell minimization will now only set defaults for maximum iterations or function evaluations if neither limit is set by the caller. In some cases with a slow converging function and only 1 limit set, the minimization may continue for longer than with previous versions and so is more likely to reach convergence. See issue gh-5966.

`scipy.stats` improvements

Trapezoidal distribution has been implemented as `scipy.stats.trapz`. Skew normal distribution has been implemented as `scipy.stats.skewnorm`. Burr type XII distribution has been implemented as `scipy.stats.burr12`. Three- and four-parameter kappa distributions have been implemented as `scipy.stats.kappa3` and `scipy.stats.kappa4`, respectively.

New `scipy.stats.iqr` function computes the interquartile region of a distribution.

Random matrices

scipy.stats.special_ortho_group and *scipy.stats.ortho_group* provide generators of random matrices in the SO(N) and O(N) groups, respectively. They generate matrices in the Haar distribution, the only uniform distribution on these group manifolds.

scipy.stats.random_correlation provides a generator for random correlation matrices, given specified eigenvalues.

`scipy.linalg` improvements

scipy.linalg.svd gained a new keyword argument, `lapack_driver`. Available drivers are `gesdd` (default) and `gesvd`.

scipy.linalg.lapack.ilaver returns the version of the LAPACK library SciPy links to.

`scipy.spatial` improvements

Boolean distances, *scipy.spatial.pdist*, have been sped up. Improvements vary by the function and the input size. In many cases, one can expect a speed-up of x2–x10.

New class *scipy.spatial.SphericalVoronoi* constructs Voronoi diagrams on the surface of a sphere. See pull request gh-5232 for details.

`scipy.cluster` improvements

A new clustering algorithm, the nearest neighbor chain algorithm, has been implemented for *scipy.cluster.hierarchy.linkage*. As a result, one can expect a significant algorithmic improvement ($O(N^2)$ instead of $O(N^3)$) for several linkage methods.

`scipy.special` improvements

The new function *scipy.special.loggamma* computes the principal branch of the logarithm of the Gamma function. For real input, `loggamma` is compatible with *scipy.special.gammaln*. For complex input, it has more consistent behavior in the complex plane and should be preferred over `gammaln`.

Vectorized forms of spherical Bessel functions have been implemented as *scipy.special.spherical_jn*, *scipy.special.spherical_kn*, *scipy.special.spherical_in* and *scipy.special.spherical_yn*. They are recommended for use over `sph_*` functions, which are now deprecated.

Several special functions have been extended to the complex domain and/or have seen domain/stability improvements. This includes *spence*, *digamma*, *log1p* and several others.

1.4.2 Deprecated features

The cross-class properties of *lti* systems have been deprecated. The following properties/setters will raise a *DeprecationWarning*:

Name - (accessing/setting raises warning) - (setting raises warning) * StateSpace - (*num*, *den*, *gain*) - (*zeros*, *poles*) * TransferFunction (*A*, *B*, *C*, *D*, *gain*) - (*zeros*, *poles*) * ZerosPolesGain (*A*, *B*, *C*, *D*, *num*, *den*) - ()

Spherical Bessel functions, `sph_in`, `sph_jn`, `sph_kn`, `sph_yn`, `sph_jnyn` and `sph_inkn` have been deprecated in favor of *scipy.special.spherical_jn* and `spherical_kn`, `spherical_yn`, `spherical_in`.

The following functions in *scipy.constants* are deprecated: `C2K`, `K2C`, `C2F`, `F2C`, `F2K` and `K2F`. They are superceded by a new function *scipy.constants.convert_temperature* that can perform all those conversions plus to/from the Rankine temperature scale.

1.4.3 Backwards incompatible changes

scipy.optimize

The convergence criterion for `optimize.bisect`, `optimize.brentq`, `optimize.brenth`, and `optimize.ridder` now works the same as `numpy.allclose`.

scipy.ndimage

The offset in `ndimage.iterpolation.affine_transform` is now consistently added after the matrix is applied, independent of if the matrix is specified using a one-dimensional or a two-dimensional array.

scipy.stats

`stats.ks_2samp` used to return nonsensical values if the input was not real or contained nans. It now raises an exception for such inputs.

Several deprecated methods of *scipy.stats* distributions have been removed: `est_loc_scale`, `vecfunc`, `veccdf` and `vec_generic_moment`.

Deprecated functions `nanmean`, `nanstd` and `nanmedian` have been removed from *scipy.stats*. These functions were deprecated in scipy 0.15.0 in favor of their `numpy` equivalents.

A bug in the `rvs()` method of the distributions in *scipy.stats* has been fixed. When arguments to `rvs()` were given that were shaped for broadcasting, in many cases the returned random samples were not random. A simple example of the problem is `stats.norm.rvs(loc=np.zeros(10))`. Because of the bug, that call would return 10 identical values. The bug only affected code that relied on the broadcasting of the shape, location and scale parameters.

The `rvs()` method also accepted some arguments that it should not have. There is a potential for backwards incompatibility in cases where `rvs()` accepted arguments that are not, in fact, compatible with broadcasting. An example is

> stats.gamma.rvs([2, 5, 10, 15], size=(2,2))

The shape of the first argument is not compatible with the requested size, but the function still returned an array with shape (2, 2). In scipy 0.18, that call generates a `ValueError`.

scipy.io

scipy.io.netcdf masking now gives precedence to the `_FillValue` attribute over the `missing_value` attribute, if both are given. Also, data are only treated as missing if they match one of these attributes exactly: values that differ by roundoff from `_FillValue` or `missing_value` are no longer treated as missing values.

scipy.interpolate

scipy.interpolate.PiecewisePolynomial class has been removed. It has been deprecated in scipy 0.14.0, and *scipy.interpolate.BPoly.from_derivatives* serves as a drop-in replacement.

1.4.4 Other changes

Scipy now uses `setuptools` for its builds instead of plain distutils. This fixes usage of `install_requires='scipy'` in the `setup.py` files of projects that depend on Scipy (see Numpy issue gh-6551 for details). It potentially affects the way that build/install methods for Scipy itself behave though. Please report any unexpected behavior on the Scipy issue tracker.

PR #6240 changes the interpretation of the *maxfun* option in *L-BFGS-B* based routines in the `scipy.optimize` module. An *L-BFGS-B* search consists of multiple iterations, with each iteration consisting of one or more function evaluations. Whereas the old search strategy terminated immediately upon reaching *maxfun* function evaluations, the new strategy allows the current iteration to finish despite reaching *maxfun*.

The bundled copy of Qhull in the `scipy.spatial` subpackage has been upgraded to version 2015.2.

The bundled copy of ARPACK in the `scipy.sparse.linalg` subpackage has been upgraded to arpack-ng 3.3.0.

The bundled copy of SuperLU in the `scipy.sparse` subpackage has been upgraded to version 5.1.1.

1.4.5 Authors

- @endolith
- @yanxun827 +
- @kleskjr +
- @MYheavyGo +
- @solarjoe +
- Gregory Allen +
- Gilles Aouizerate +
- Tom Augspurger +
- Henrik Bengtsson +
- Felix Berkenkamp
- Per Brodtkorb
- Lars Buitinck
- Daniel Bunting +
- Evgeni Burovski
- CJ Carey
- Tim Cera
- Grey Christoforo +
- Robert Cimrman
- Philip DeBoer +
- Yves Delley +
- Dávid Bodnár +
- Ion Elberdin +
- Gabriele Farina +

- Yu Feng
- Andrew Fowlie +
- Joseph Fox-Rabinovitz
- Simon Gibbons +
- Neil Girdhar +
- Kolja Glogowski +
- Christoph Gohlke
- Ralf Gommers
- Todd Goodall +
- Johnnie Gray +
- Alex Griffing
- Olivier Grisel
- Thomas Haslwanter +
- Michael Hirsch +
- Derek Homeier
- Golnaz Irannejad +
- Marek Jacob +
- InSuk Joung +
- Tetsuo Koyama +
- Eugene Krokhalev +
- Eric Larson
- Denis Laxalde
- Antony Lee
- Jerry Li +
- Henry Lin +
- Nelson Liu +
- Loïc Estève
- Lei Ma +
- Osvaldo Martin +
- Stefano Martina +
- Nikolay Mayorov
- Matthieu Melot +
- Sturla Molden
- Eric Moore
- Alistair Muldal +
- Maniteja Nandana

- Tavi Nathanson +
- Andrew Nelson
- Joel Nothman
- Behzad Nouri
- Nikolai Nowaczyk +
- Juan Nunez-Iglesias +
- Ted Pudlik
- Eric Quintero
- Yoav Ram
- Jonas Rauber +
- Tyler Reddy +
- Juha Remes
- Garrett Reynolds +
- Ariel Rokem +
- Fabian Rost +
- Bill Sacks +
- Jona Sassenhagen +
- Kari Schoonbee +
- Marcello Seri +
- Sourav Singh +
- Martin Spacek +
- Søren Fuglede Jørgensen +
- Bhavika Tekwani +
- Martin Thoma +
- Sam Tygier +
- Meet Udeshi +
- Utkarsh Upadhyay
- Bram Vandekerckhove +
- Sebastián Vanrell +
- Ze Vinicius +
- Pauli Virtanen
- Stefan van der Walt
- Warren Weckesser
- Jakub Wilk +
- Josh Wilson
- Phillip J. Wolfram +

- Nathan Woods
- Haochen Wu
- G Young +

A total of 99 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.18.0

- #1484: SVD using `*GESVD` lapack drivers (Trac #957)
- #1547: Inconsistent use of offset in ndimage.interpolation.affine_transform()...
- #1609: special.hyp0f1 returns nan (Trac #1082)
- #1656: fmin_slsqp enhancement (Trac #1129)
- #2069: stats broadcasting in rvs (Trac #1544)
- #2165: sph_jn returns false results for some orders/values (Trac #1640)
- #2255: Incorrect order of translation and rotation in affine_transform...
- #2332: hyp0f1 args and return values are unnumpyic (Trac #1813)
- #2534: The sparse .sum() method with uint8 dtype does not act like the...
- #3113: Implement ufuncs for CSPHJY, SPHJ, SPHY, CSPHIK, SPHI, SPHIK...
- #3568: SciPy 0.13.3 - CentOS5 - Errors in test_arpack
- #3581: optimize: stepsize in fmin_bfgs is "bad"
- #4476: scipy.sparse non-native endian bug
- #4484: ftol in optimize.fmin fails to work
- #4510: sparsetools.cxx call_thunk can segfault due to out of bounds...
- #5051: ftol and xtol for _minimize_neldermead are absolute instead of...
- #5097: proposal: spherical Voronoi diagrams
- #5123: Call to *scipy.sparse.coo_matrix* fails when passed Cython typed...
- #5220: scipy.cluster.hierarchy.{ward,median,centroid} does not work...
- #5379: Add a build step at the end of .travis.yml that uploads working...
- #5440: scipy.optimize.basinhopping: accept_test returning `numpy.bool_`...
- #5452: Error in scipy.integrate.nquad when using variable integration...
- #5520: Cannot inherit csr_matrix properly
- #5533: Kendall tau implementation uses Python mergesort
- #5553: stats.tiecorrect overflows
- #5589: Add the Type XII Burr distribution to stats.
- #5612: sparse.linalg factorizations slow for small k due to default...
- #5626: io.netcdf masking should use masked_equal rather than masked_value
- #5637: Simple cubic spline interpolation?

- #5683: BUG: Akima1DInterpolator may return nans given multidimensional...
- #5686: scipy.stats.ttest_ind_from_stats does not accept arrays
- #5702: scipy.ndimage.interpolation.affine_transform lacks documentation...
- #5718: Wrong computation of weighted minkowski distance in cdist
- #5745: move to setuptools for next release
- #5752: DOC: solve_discrete_lyapunov equation puts transpose in wrong...
- #5760: signal.ss2tf doesn't handle zero-order state-space models
- #5764: Hypergeometric function hyp0f1 behaves incorrectly for complex...
- #5814: stats NaN Policy Error message inconsistent with code
- #5833: docstring of stats.binom_test() needs an update
- #5853: Error in scipy.linalg.expm for complex matrix with shape (1,1)
- #5856: Specify Nelder-Mead initial simplex
- #5865: scipy.linalg.expm fails for certain numpy matrices
- #5915: optimize.basinhopping - variable referenced before assignment.
- #5916: LSQUnivariateSpline fitting failed with knots generated from...
- #5927: unicode vs. string comparison in scipy.stats.binned_statistic_dd
- #5936: faster implementation of ks_2samp
- #5948: csc matrix .mean returns single element matrix rather than scalar
- #5959: BUG: optimize test error for root when using lgmres
- #5972: Test failures for sparse sum tests on 32-bit Python
- #5976: Unexpected exception in scipy.sparse.bmat while using 0 x 0 matrix
- #6008: scipy.special.kl_div not available in 0.14.1
- #6011: The von-Mises entropy is broken
- #6016: python crashes for linalg.interpolative.svd with certain large...
- #6017: Wilcoxon signed-rank test with zero_method="pratt" or "zsplit"...
- #6028: stats.distributions does not have trapezoidal distribution
- #6035: Wrong link in f_oneway
- #6056: BUG: signal.decimate should only accept discrete LTI objects
- #6093: Precision error on Linux 32 bit with openblas
- #6101: Barycentric transforms test error on Python3, 32-bit Linux
- #6105: scipy.misc.face docstring is incorrect
- #6113: scipy.linalg.logm fails for a trivial matrix
- #6128: Error in dot method of sparse COO array, when used with numpy...
- #6132: Failures with latest MKL
- #6136: Failures on *master* with MKL
- #6162: fmin_l_bfgs_b returns inconsistent results (fmin f(xmin)) and...

- #6165: optimize.minimize infinite loop with Newton-CG
- #6167: incorrect distribution fitting for data containing boundary values.
- #6194: lstsq() and others detect numpy.complex256 as real
- #6216: ENH: improve accuracy of ppf cdf roundtrip for bradford
- #6217: BUG: weibull_min.logpdf return nan for c=1 and x=0
- #6218: Is there a method to cap shortest path search distances?
- #6222: PchipInterpolator no longer handles a 2-element array
- #6226: ENH: improve accuracy for logistic.ppf and logistic.isf
- #6227: ENH: improve accuracy for rayleigh.logpdf and rayleigh.logsf...
- #6228: ENH: improve accuracy of ppf cdf roundtrip for gumbel_l
- #6235: BUG: alpha.pdf and alpha.logpdf returns nan for x=0
- #6245: ENH: improve accuracy for ppf-cdf and sf-isf roundtrips for invgamma
- #6263: BUG: stats: Inconsistency in the multivariate_normal docstring
- #6292: Python 3 unorderable type errors in test_sparsetools.TestInt32Overflow
- #6316: TestCloughTocher2DInterpolator.test_dense crashes python3.5.2rc1_64bit...
- #6318: Scipy interp1d 'nearest' not working for high values on x-axis

Pull requests for 0.18.0

- #3226: DOC: Change *nb* and *na* to conventional m and n
- #3867: allow cKDTree.query taking a list input in k.
- #4191: ENH: Benchmarking global optimizers
- #4356: ENH: add PPoly.solve(y) for solving `p(x) == y`
- #4370: DOC separate boolean distance functions for clarity
- #4678: BUG: sparse: ensure index dtype is large enough to pass all parameters...
- #4881: scipy.signal: Add the class dlti for linear discrete-time systems....
- #4901: MAINT: add benchmark and improve docstring for signal.lfilter
- #5043: ENH: sparse: add count_nonzero method
- #5136: Attribute kurtosistest() to Anscombe & Glynn (1983)
- #5186: ENH: Port upfirdn
- #5232: ENH: adding spherical Voronoi diagram algorithm to scipy.spatial
- #5279: ENH: Bessel filters with different normalizations, high order
- #5384: BUG: Closes #5027 distance function always casts bool to double
- #5392: ENH: Add zero_phase kwarg to signal.decimate
- #5394: MAINT: sparse: non-canonical test cleanup and fixes
- #5424: DOC: add Scipy developers guide
- #5442: STY: PEP8 amendments

- #5472: Online QR in LGMRES
- #5526: BUG: stats: Fix broadcasting in the rvs() method of the distributions.
- #5530: MAINT: sparse: set *format* attr explicitly
- #5536: optimize: fix up cg/bfgs initial step sizes
- #5548: PERF: improves performance in stats.kendalltau
- #5549: ENH: Nearest-neighbor chain algorithm for hierarchical clustering
- #5554: MAINT/BUG: closes overflow bug in stats.tiecorrect
- #5557: BUG: modify optimize.bisect to achieve desired tolerance
- #5581: DOC: Tutorial for least_squares
- #5606: ENH: differential_evolution - moving core loop of solve method...
- #5609: [MRG] test against numpy dev
- #5611: use setuptools for bdist_egg distributions
- #5615: MAINT: linalg: tighten _decomp_update + special: remove unused...
- #5622: Add SO(N) rotation matrix generator
- #5623: ENH: special: Add vectorized spherical Bessel functions.
- #5627: Response to issue #5160, implements the skew normal distribution...
- #5628: DOC: Align the description and operation
- #5632: DOC: special: Expanded docs for Airy, elliptic, Bessel functions.
- #5633: MAINT: linalg: unchecked malloc in _decomp_update
- #5634: MAINT: optimize: tighten _group_columns
- #5640: Fixes for io.netcdf masking
- #5645: MAINT: size 0 vector handling in cKDTree range queries
- #5649: MAINT: update license text
- #5650: DOC: Clarify Exponent Order in ltisys.py
- #5651: DOC: Clarify Documentation for scipy.special.gammaln
- #5652: DOC: Fixed scipy.special.betaln Doc
- #5653: [MRG] ENH: CubicSpline interpolator
- #5654: ENH: Burr12 distribution to stats module
- #5659: DOC: Define BEFORE/AFTER in runtests.py -h for bench-compare
- #5660: MAINT: remove functions deprecated before 0.16.0
- #5662: ENH: Circular statistic optimization
- #5663: MAINT: remove uses of np.testing.rand
- #5665: MAINT: spatial: remove matching distance implementation
- #5667: Change some HTTP links to HTTPS
- #5669: DOC: zpk2sos can't do analog, array_like, etc.
- #5670: Update conf.py

- #5672: MAINT: move a sample distribution to a subclass of rv_discrete
- #5678: MAINT: stats: remove est_loc_scale method
- #5679: MAINT: DRY up generic computations for discrete distributions
- #5680: MAINT: stop shadowing builtins in stats.distributions
- #5681: forward port ENH: Re-enable broadcasting of fill_value
- #5684: BUG: Fix Akima1DInterpolator returning nans
- #5690: BUG: fix stats.ttest_ind_from_stats to handle arrays.
- #5691: BUG: fix generator in io._loadarff to comply with PEP 0479
- #5693: ENH: use math.factorial for exact factorials
- #5695: DOC: dx might be a float, not only an integer
- #5699: MAINT: io: micro-optimize Matlab reading code for size
- #5701: Implement OptimizeResult.__dir__
- #5703: ENH: stats: make R^2 printing optional in probplot
- #5704: MAINT: typo ouf->out
- #5705: BUG: fix typo in query_pairs
- #5707: DOC:Add some explanation for ftol xtol in scipy.optimize.fmin
- #5708: DOC: optimize: PEP8 minimize docstring
- #5709: MAINT: optimize Cython code for speed and size
- #5713: [DOC] Fix broken link to reference
- #5717: DOC: curve_fit raises RuntimeError on failure.
- #5724: forward port gh-5720
- #5728: STY: remove a blank line
- #5729: ENH: spatial: speed up boolean distances
- #5732: MAINT: differential_evolution changes to default keywords break...
- #5733: TST: differential_evolution - population initiation tests
- #5736: Complex number support in log1p, expm1, and xlog1py
- #5741: MAINT: sparse: clean up extraction functions
- #5742: DOC: signal: Explain fftbins in get_window
- #5748: ENH: Add O(N) random matrix generator
- #5749: ENH: Add polyphase resampling
- #5756: RFC: Bump the minimum numpy version, drop older python versions
- #5761: DOC: Some improvements to least squares docstrings
- #5762: MAINT: spatial: distance refactoring
- #5768: DOC: Fix io.loadmat docstring for mdict param
- #5770: BUG: Accept anything np.dtype can handle for a dtype in sparse.random
- #5772: Update sparse.csgraph.laplacian docstring

- #5777: BUG: fix special.hyp0f1 to work correctly for complex inputs.
- #5780: DOC: Update PIL error install URL
- #5781: DOC: Fix documentation on solve_discrete_lyapunov
- #5782: DOC: cKDTree and KDTree now reference each other
- #5783: DOC: Clarify finish behaviour in scipy.optimize.brute
- #5784: MAINT: Change default tolerances of least_squares to 1e-8
- #5787: BUG: Allow Processing of Zero Order State Space Models in signal.ss2tf
- #5788: DOC, BUG: Clarify and Enforce Input Types to 'Data' Objects
- #5789: ENH: sparse: speedup LIL matrix slicing (was #3338)
- #5791: DOC: README: remove coveralls.io
- #5792: MAINT: remove uses of deprecated np.random.random_integers
- #5794: fix affine_transform (fixes #1547 and #5702)
- #5795: DOC: Removed uniform method from kmeans2 doc
- #5797: DOC: Clarify the computation of weighted minkowski
- #5798: BUG: Ensure scipy's _asfarray returns ndarray
- #5799: TST: Mpmath testing patch
- #5801: allow reading of certain IDL 8.0 .sav files
- #5803: DOC: fix module name in error message
- #5804: DOC: special: Expanded docs for special functions.
- #5805: DOC: Fix order of returns in _spectral_helper
- #5806: ENH: sparse: vectorized coo_matrix.diagonal
- #5808: ENH: Added iqr function to compute IQR metric in scipy/stats/stats.py
- #5810: MAINT/BENCH: sparse: Benchmark cleanup and additions
- #5811: DOC: sparse.linalg: shape, not size
- #5813: Update sparse ARPACK functions min *ncv* value
- #5815: BUG: Error message contained wrong values
- #5816: remove dead code from stats tests
- #5820: "in"->"a" in order_filter docstring
- #5821: DOC: README: INSTALL.txt was renamed in 2014
- #5825: DOC: typo in the docstring of least_squares
- #5826: MAINT: sparse: increase test coverage
- #5827: NdPPoly rebase
- #5828: Improve numerical stability of hyp0f1 for large orders
- #5829: ENH: sparse: Add copy parameter to all .toXXX() methods in sparse...
- #5830: DOC: rework INSTALL.rst.txt
- #5831: Adds plotting options to voronoi_plot_2d

- #5834: Update stats.binom_test() docstring
- #5836: ENH, TST: Allow SIMO tf's for tf2ss
- #5837: DOC: Image examples
- #5838: ENH: sparse: add eliminate_zeros() to coo_matrix
- #5839: BUG: Fixed name of NumpyVersion.__repr__
- #5845: MAINT: Fixed typos in documentation
- #5847: Fix bugs in sparsetools
- #5848: BUG: sparse.linalg: add locks to ensure ARPACK threadsafety
- #5849: ENH: sparse.linalg: upgrade to superlu 5.1.1
- #5851: ENH: expose lapack's ilaver to python to allow lapack verion...
- #5852: MAINT: runtests.py: ensure Ctrl-C interrupts the build
- #5854: DOC: Minor update to documentation
- #5855: Pr 5640
- #5859: ENH: Add random correlation matrix generator
- #5862: BUG: Allow expm for complex matrix with shape (1, 1)
- #5863: FIX: Fix test
- #5864: DOC: add a little note about the Normal survival function (Q-function)
- #5867: Fix for #5865
- #5869: extend normal distribution cdf to complex domain
- #5872: DOC: Note that morlet and cwt don't work together
- #5875: DOC: interp2d class description
- #5876: MAINT: spatial: remove a stray print statement
- #5878: MAINT: Fixed noisy UserWarnings in ndimage tests. Fixes #5877
- #5879: MAINT: sparse.linalg/superlu: add explicit casts to resolve compiler...
- #5880: MAINT: signal: import gcd from math and not fractions when on...
- #5887: Neldermead initial simplex
- #5894: BUG: _CustomLinearOperator unpickalable in python3.5
- #5895: DOC: special: slightly improve the multigammaln docstring
- #5900: Remove duplicate assignment.
- #5901: Update bundled ARPACK
- #5904: ENH: Make convolve and correlate order-agnostic
- #5905: ENH: sparse.linalg: further LGMRES cleanups
- #5906: Enhancements and cleanup in scipy.integrate (attempt #2)
- #5907: ENH: Change sparse *sum* and *mean* dtype casting to match...
- #5909: changes for convolution symmetry
- #5913: MAINT: basinhopping remove instance test closes #5440

- #5919: MAINT: uninitialised var if basinhopping niter=0. closes #5915
- #5920: BLD: Fix missing lsame.c error for MKL
- #5921: DOC: interpolate: add example showing how to work around issue...
- #5926: MAINT: spatial: upgrade to Qhull 2015.2
- #5928: MAINT: sparse: optimize DIA sum/diagonal, csgraph.laplacian
- #5929: Update info/URL for octave-maintainers discussion
- #5930: TST: special: silence DeprecationWarnings from sph_yn
- #5931: ENH: implement the principle branch of the logarithm of Gamma.
- #5934: Typo: "mush" => "must"
- #5935: BUG:string comparison stats._binned_statistic closes #5927
- #5938: Cythonize stats.ks_2samp for a ~33% gain in speed.
- #5939: DOC: fix optimize.fmin convergence docstring
- #5941: Fix minor typo in squareform docstring
- #5942: Update linregress stderr description.
- #5943: ENH: Improve numerical accuracy of lognorm
- #5944: Merge vonmises into stats pyx
- #5945: MAINT: interpolate: Tweak declaration to avoid cython warning...
- #5946: MAINT: sparse: clean up format conversion methods
- #5949: BUG: fix sparse .mean to return a scalar instead of a matrix
- #5955: MAINT: Replace calls to *hanning* with *hann*
- #5956: DOC: Missing periods interfering with parsing
- #5958: MAINT: add a test for lognorm.sf underflow
- #5961: MAINT _centered(): rename size to shape
- #5962: ENH: constants: Add multi-scale temperature conversion function
- #5965: ENH: special: faster way for calculating comb() for exact=True
- #5975: ENH: Improve FIR path of signal.decimate
- #5977: MAINT/BUG: sparse: remove overzealous bmat checks
- #5978: minimize_neldermead() stop at user requested maxiter or maxfev
- #5983: ENH: make sparse *sum* cast dtypes like NumPy *sum* for 32-bit...
- #5985: BUG, API: Add *jac* parameter to curve_fit
- #5989: ENH: Add firls least-squares fitting
- #5990: BUG: read tries to handle 20-bit WAV files but shouldn't
- #5991: DOC: Cleanup wav read/write docs and add tables for common types
- #5994: ENH: Add gesvd method for svd
- #5996: MAINT: Wave cleanup
- #5997: TST: Break up upfirdn tests & compare to lfilter

- #6001: Filter design docs
- #6002: COMPAT: Expand compatibility fromnumeric.py
- #6007: ENH: Skip conversion of TF to TF in freqresp
- #6009: DOC: fix incorrect versionadded for entr, rel_entr, kl_div
- #6013: Fixed the entropy calculation of the von Mises distribution.
- #6014: MAINT: make gamma, rgamma use loggamma for complex arguments
- #6020: WIP: ENH: add exact=True factorial for vectors
- #6022: Added 'lanczos' to the image interpolation function list.
- #6024: BUG: optimize: do not use dummy constraints in SLSQP when no...
- #6025: ENH: Boundary value problem solver for ODE systems
- #6029: MAINT: Future imports for optimize._lsq
- #6030: ENH: stats.trap - adding trapezoidal distribution closes #6028
- #6031: MAINT: Some improvements to optimize._numdiff
- #6032: MAINT: Add special/_comb.c to .gitignore
- #6033: BUG: check the requested approximation rank in interpolative.svd
- #6034: DOC: Doc for mannwhitneyu in stats.py corrected
- #6040: FIX: Edit the wrong link in f_oneway
- #6044: BUG: (ordqz) always increase parameter lwork by 1.
- #6047: ENH: extend special.spence to complex arguments.
- #6049: DOC: Add documentation of PR #5640 to the 0.18.0 release notes
- #6050: MAINT: small cleanups related to loggamma
- #6070: Add asarray to explicitly cast list to numpy array in wilcoxon...
- #6071: DOC: antialiasing filter and link decimate resample, etc.
- #6075: MAINT: reimplement special.digamma for complex arguments
- #6080: avoid multiple computation in kstest
- #6081: Clarified pearson correlation return value
- #6085: ENH: allow long indices of sparse matrix with umfpack in spsolve()
- #6086: fix description for associated Laguerre polynomials
- #6087: Corrected docstring of splrep.
- #6094: ENH: special: change zeta signature to zeta(x, q=1)
- #6095: BUG: fix integer overflow in special.spence
- #6106: Fixed Issue #6105
- #6116: BUG: matrix logarithm edge case
- #6119: TST: DeprecationWarnings in stats on python 3.5 closes #5885
- #6120: MAINT: sparse: clean up sputils.isintlike
- #6122: DOC: optimize: linprog docs should say minimize instead of maximize

- #6123: DOC: optimize: document the *fun* field in `scipy.optimize.OptimizeResult`
- #6124: Move FFT zero-padding calculation from signaltools to fftpack
- #6125: MAINT: improve special.gammainc in the `a ~ x` regime.
- #6130: BUG: sparse: Fix COO dot with zero columns
- #6138: ENH: stats: Improve behavior of genextreme.sf and genextreme.isf
- #6146: MAINT: simplify the expit implementation
- #6151: MAINT: special: make generate_ufuncs.py output deterministic
- #6152: TST: special: better test for gammainc at large arguments
- #6153: ENH: Make next_fast_len public and faster
- #6154: fix typo “mush”–>”must”
- #6155: DOC: Fix some incorrect RST definition lists
- #6160: make logsumexp error out on a masked array
- #6161: added missing bracket to rosen documentation
- #6163: ENH: Added “kappa4” and “kappa3” distributions.
- #6164: DOC: Minor clean-up in integrate._bvp
- #6169: Fix mpf_assert_allclose to handle iterable results, such as maps
- #6170: Fix pchip_interpolate convenience function
- #6172: Corrected misplaced bracket in doc string
- #6175: ENH: sparse.csgraph: Pass indices to shortest_path
- #6178: TST: increase test coverage of sf and isf of a generalized extreme...
- #6179: TST: avoid a deprecation warning from numpy
- #6181: ENH: Boundary conditions for CubicSpline
- #6182: DOC: Add examples/graphs to max_len_seq
- #6183: BLD: update Bento build config files for recent changes.
- #6184: BUG: fix issue in io/wavfile for float96 input.
- #6186: ENH: Periodic extrapolation for PPoly and BPoly
- #6192: MRG: Add circle-CI
- #6193: ENH: sparse: avoid setitem densification
- #6196: Fixed missing sqrt in docstring of Mahalanobis distance in cdist,...
- #6206: MAINT: Minor changes in solve_bvp
- #6207: BUG: linalg: for BLAS, downcast complex256 to complex128, not...
- #6209: BUG: io.matlab: avoid buffer overflows in read_element_into
- #6210: BLD: use setuptools when building.
- #6214: BUG: sparse.linalg: fix bug in LGMRES breakdown handling
- #6215: MAINT: special: make loggamma use zdiv
- #6220: DOC: Add parameter

- #6221: ENH: Improve Newton solver for solve_bvp
- #6223: pchip should work for length-2 arrays
- #6224: signal.lti: deprecate cross-class properties/setters
- #6229: BUG: optimize: avoid an infinite loop in Newton-CG
- #6230: Add example for application of gaussian filter
- #6236: MAINT: gumbel_l accuracy
- #6237: MAINT: rayleigh accuracy
- #6238: MAINT: logistic accuracy
- #6239: MAINT: bradford distribution accuracy
- #6240: MAINT: avoid bad fmin in l-bfgs-b due to maxfun interruption
- #6241: MAINT: weibull_min accuracy
- #6246: ENH: Add _support_mask to distributions
- #6247: fixed a print error for an example of ode
- #6249: MAINT: change x-axis label for stats.probplot to "theoretical...
- #6250: DOC: fix typos
- #6251: MAINT: constants: filter out test noise from deprecated conversions
- #6252: MAINT: io/arff: remove unused variable
- #6253: Add examples to scipy.ndimage.filters
- #6254: MAINT: special: fix some build warnings
- #6258: MAINT: inverse gamma distribution accuracy
- #6260: MAINT: signal.decimate - Use discrete-time objects
- #6262: BUG: odr: fix string formatting
- #6267: TST: fix some test issues in interpolate and stats.
- #6269: TST: fix some warnings in the test suite
- #6274: ENH: Add sosfiltfilt
- #6276: DOC: update release notes for 0.18.0
- #6277: MAINT: update the author name mapping
- #6282: DOC: Correcting references for scipy.stats.normaltest
- #6283: DOC: some more additions to 0.18.0 release notes.
- #6284: Add *versionadded::* directive to *loggamma*.
- #6285: BUG: stats: Inconsistency in the multivariate_normal docstring...
- #6290: Add author list, gh-lists to 0.18.0 release notes
- #6293: TST: special: relax a test's precision
- #6295: BUG: sparse: stop comparing None and int in bsr_matrix constructor
- #6313: MAINT: Fix for python 3.5 travis-ci build problem.
- #6327: TST: signal: use assert_allclose for testing near-equality in...

- #6330: BUG: spatial/qhull: allocate qhT via malloc to ensure CRT likes...
- #6332: TST: fix stats.iqr test to not emit warnings, and fix line lengths.
- #6334: MAINT: special: fix a test for hyp0f1
- #6347: TST: spatial.qhull: skip a test on 32-bit platforms
- #6350: BUG: optimize/slsqp: don't overwrite an array out of bounds
- #6351: BUG: #6318 Interp1d 'nearest' integer x-axis overflow issue fixed
- #6355: Backports for 0.18.0

1.5 SciPy 0.17.1 Release Notes

SciPy 0.17.1 is a bug-fix release with no new features compared to 0.17.0.

1.5.1 Issues closed for 0.17.1

- #5817: BUG: skew, kurtosis return np.nan instead of "propagate"
- #5850: Test failed with sgelsy
- #5898: interpolate.interp1d crashes using float128
- #5953: Massive performance regression in cKDTree.query with L_inf distance...
- #6062: mannwhitneyu breaks backward compatibility in 0.17.0
- #6134: T test does not handle nans

1.5.2 Pull requests for 0.17.1

- #5902: BUG: interpolate: make interp1d handle np.float128 again
- #5957: BUG: slow down with p=np.inf in 0.17 cKDTree.query
- #5970: Actually propagate nans through stats functions with nan_policy="propagate"
- #5971: BUG: linalg: fix lwork check in `*gelsy`
- #6074: BUG: special: fixed violation of strict aliasing rules.
- #6083: BUG: Fix dtype for sum of linear operators
- #6100: BUG: Fix mannwhitneyu to be backward compatible
- #6135: Don't pass null pointers to LAPACK, even during workspace queries.
- #6148: stats: fix handling of nan values in T tests and kendalltau

1.6 SciPy 0.17.0 Release Notes

Contents

- *SciPy 0.17.0 Release Notes*

- *New features*
  - `scipy.cluster` *improvements*
  - `scipy.io` *improvements*
  - `scipy.optimize` *improvements*
    - *Linear assignment problem solver*
    - *Least squares optimization*
  - `scipy.signal` *improvements*
  - `scipy.stats` *improvements*
  - `scipy.sparse` *improvements*
  - `scipy.spatial` *improvements*
  - `scipy.interpolate` *improvements*
  - `scipy.linalg` *improvements*
- *Deprecated features*
- *Backwards incompatible changes*
- *Other changes*
- *Authors*
  - *Issues closed for 0.17.0*
  - *Pull requests for 0.17.0*

SciPy 0.17.0 is the culmination of 6 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.17.x branch, and on adding new features on the master branch.

This release requires Python 2.6, 2.7 or 3.2-3.5 and NumPy 1.6.2 or greater.

Release highlights:

- New functions for linear and nonlinear least squares optimization with constraints: *scipy.optimize.lsq_linear* and *scipy.optimize.least_squares*
- Support for fitting with bounds in *scipy.optimize.curve_fit*.
- Significant improvements to *scipy.stats*, providing many functions with better handing of inputs which have NaNs or are empty, improved documentation, and consistent behavior between *scipy.stats* and *scipy.stats.mstats*.
- Significant performance improvements and new functionality in *scipy.spatial.cKDTree*.

1.6.1 New features

`scipy.cluster` improvements

A new function *scipy.cluster.hierarchy.cut_tree*, which determines a cut tree from a linkage matrix, was added.

scipy.io improvements

scipy.io.mmwrite gained support for symmetric sparse matrices.

scipy.io.netcdf gained support for masking and scaling data based on data attributes.

scipy.optimize improvements

Linear assignment problem solver

scipy.optimize.linear_sum_assignment is a new function for solving the linear sum assignment problem. It uses the Hungarian algorithm (Kuhn-Munkres).

Least squares optimization

A new function for *nonlinear* least squares optimization with constraints was added: *scipy.optimize.least_squares*. It provides several methods: Levenberg-Marquardt for unconstrained problems, and two trust-region methods for constrained ones. Furthermore it provides different loss functions. New trust-region methods also handle sparse Jacobians.

A new function for *linear* least squares optimization with constraints was added: *scipy.optimize.lsq_linear*. It provides a trust-region method as well as an implementation of the Bounded-Variable Least-Squares (BVLS) algorithm.

scipy.optimize.curve_fit now supports fitting with bounds.

scipy.signal improvements

A `mode` keyword was added to *scipy.signal.spectrogram*, to let it return other spectrograms than power spectral density.

scipy.stats improvements

Many functions in *scipy.stats* have gained a `nan_policy` keyword, which allows specifying how to treat input with NaNs in them: propagate the NaNs, raise an error, or omit the NaNs.

Many functions in *scipy.stats* have been improved to correctly handle input arrays that are empty or contain infs/nans.

A number of functions with the same name in *scipy.stats* and *scipy.stats.mstats* were changed to have matching signature and behavior. See gh-5474 for details.

scipy.stats.binom_test and *scipy.stats.mannwhitneyu* gained a keyword `alternative`, which allows specifying the hypothesis to test for. Eventually all hypothesis testing functions will get this keyword.

For methods of many continuous distributions, complex input is now accepted.

Matrix normal distribution has been implemented as *scipy.stats.matrix_normal*.

scipy.sparse improvements

The *axis* keyword was added to sparse norms, *scipy.sparse.linalg.norm*.

`scipy.spatial` improvements

scipy.spatial.cKDTree was partly rewritten for improved performance and several new features were added to it:

- the `query_ball_point` method became significantly faster
- `query` and `query_ball_point` gained an `n_jobs` keyword for parallel execution
- build and query methods now release the GIL
- full pickling support
- support for periodic spaces
- the `sparse_distance_matrix` method can now return and sparse matrix type

`scipy.interpolate` improvements

Out-of-bounds behavior of *scipy.interpolate.interp1d* has been improved. Use a two-element tuple for the `fill_value` argument to specify separate fill values for input below and above the interpolation range. Linear and nearest interpolation kinds of *scipy.interpolate.interp1d* support extrapolation via the `fill_value="extrapolate"` keyword.

`fill_value` can also be set to an array-like (or a two-element tuple of array-likes for separate below and above values) so long as it broadcasts properly to the non-interpolated dimensions of an array. This was implicitly supported by previous versions of scipy, but support has now been formalized and gets compatibility-checked before use. For example, a set of `y` values to interpolate with shape `(2, 3, 5)` interpolated along the last axis (2) could accept a `fill_value` array with shape `()` (singleton), `(1,)`, `(2, 1)`, `(1, 3)`, `(3,)`, or `(2, 3)`; or it can be a 2-element tuple to specify separate below and above bounds, where each of the two tuple elements obeys proper broadcasting rules.

`scipy.linalg` improvements

The default algorithm for *scipy.linalg.leastsq* has been changed to use LAPACK's function `*gelsd`. Users wanting to get the previous behavior can use a new keyword `lapack_driver="gelss"` (allowed values are "gelss", "gelsd" and "gelsy").

`scipy.sparse` matrices and linear operators now support the matmul (`@`) operator when available (Python 3.5+). See [PEP 465](http://legacy.python.org/dev/peps/pep-0465/)

A new function *scipy.linalg.ordqz*, for QZ decomposition with reordering, has been added.

1.6.2 Deprecated features

`scipy.stats.histogram` is deprecated in favor of `np.histogram`, which is faster and provides the same functionality.

`scipy.stats.threshold` and `scipy.mstats.threshold` are deprecated in favor of `np.clip`. See issue #617 for details.

`scipy.stats.ss` is deprecated. This is a support function, not meant to be exposed to the user. Also, the name is unclear. See issue #663 for details.

`scipy.stats.square_of_sums` is deprecated. This too is a support function not meant to be exposed to the user. See issues #665 and #663 for details.

scipy.stats.f_value, scipy.stats.f_value_multivariate, scipy.stats.f_value_wilks_lambda, and scipy.mstats.f_value_wilks_lambda are deprecated. These are related to ANOVA, for which scipy.stats provides quite limited functionality and these functions are not very useful standalone. See issues #660 and #650 for details.

scipy.stats.chisqprob is deprecated. This is an alias. stats.chi2.sf should be used instead.

scipy.stats.betai is deprecated. This is an alias for special.betainc which should be used instead.

1.6.3 Backwards incompatible changes

The functions stats.trim1 and stats.trimboth now make sure the elements trimmed are the lowest and/or highest, depending on the case. Slicing without at least partial sorting was previously done, but didn't make sense for unsorted input.

When variable_names is set to an empty list, scipy.io.loadmat now correctly returns no values instead of all the contents of the MAT file.

Element-wise multiplication of sparse matrices now returns a sparse result in all cases. Previously, multiplying a sparse matrix with a dense matrix or array would return a dense matrix.

The function misc.lena has been removed due to license incompatibility.

The constructor for sparse.coo_matrix no longer accepts (None, (m,n)) to construct an all-zero matrix of shape (m,n). This functionality was deprecated since at least 2007 and was already broken in the previous SciPy release. Use coo_matrix((m,n)) instead.

The Cython wrappers in linalg.cython_lapack for the LAPACK routines *gegs, *gegv, *gelsx, *geqpf, *ggsvd, *ggsvp, *lahrd, *latzm, *tzrqf have been removed since these routines are not present in the new LAPACK 3.6.0 release. With the exception of the routines *ggsvd and *ggsvp, these were all deprecated in favor of routines that are currently present in our Cython LAPACK wrappers.

Because the LAPACK *gegv routines were removed in LAPACK 3.6.0. The corresponding Python wrappers in scipy.linalg.lapack are now deprecated and will be removed in a future release. The source files for these routines have been temporarily included as a part of scipy.linalg so that SciPy can be built against LAPACK versions that do not provide these deprecated routines.

1.6.4 Other changes

Html and pdf documentation of development versions of Scipy is now automatically rebuilt after every merged pull request.

scipy.constants is updated to the CODATA 2014 recommended values.

Usage of *scipy.fftpack* functions within Scipy has been changed in such a way that PyFFTW can easily replace *scipy.fftpack* functions (with improved performance). See gh-5295 for details.

The imread functions in *scipy.misc* and *scipy.ndimage* were unified, for which a mode argument was added to *scipy.misc.imread*. Also, bugs for 1-bit and indexed RGB image formats were fixed.

runtests.py, the development script to build and test Scipy, now allows building in parallel with --parallel.

1.6.5 Authors

- @cel4 +
- @chemelnucfin +

- @endolith
- @mamrehn +
- @tosh1ki +
- Joshua L. Adelman +
- Anne Archibald
- Hervé Audren +
- Vincent Barrielle +
- Bruno Beltran +
- Sumit Binnani +
- Joseph Jon Booker
- Olga Botvinnik +
- Michael Boyle +
- Matthew Brett
- Zaz Brown +
- Lars Buitinck
- Pete Bunch +
- Evgeni Burovski
- CJ Carey
- Ien Cheng +
- Cody +
- Jaime Fernandez del Rio
- Ales Erjavec +
- Abraham Escalante
- Yves-Rémi Van Eycke +
- Yu Feng +
- Eric Firing
- Francis T. O'Donovan +
- André Gaul
- Christoph Gohlke
- Ralf Gommers
- Alex Griffing
- Alexander Grigorievskiy
- Charles Harris
- Jörn Hees +
- Ian Henriksen
- Derek Homeier +

- David Menéndez Hurtado
- Gert-Ludwig Ingold
- Aakash Jain +
- Rohit Jamuar +
- Jan Schlüter
- Johannes Ballé
- Luke Zoltan Kelley +
- Jason King +
- Andreas Kopecky +
- Eric Larson
- Denis Laxalde
- Antony Lee
- Gregory R. Lee
- Josh Levy-Kramer +
- Sam Lewis +
- François Magimel +
- Martín Gaitán +
- Sam Mason +
- Andreas Mayer
- Nikolay Mayorov
- Damon McDougall +
- Robert McGibbon
- Sturla Molden
- Will Monroe +
- Eric Moore
- Maniteja Nandana
- Vikram Natarajan +
- Andrew Nelson
- Marti Nito +
- Behzad Nouri +
- Daisuke Oyama +
- Giorgio Patrini +
- Fabian Paul +
- Christoph Paulik +
- Mad Physicist +
- Irvin Probst

- Sebastian Pucilowski +
- Ted Pudlik +
- Eric Quintero
- Yoav Ram +
- Joscha Reimer +
- Juha Remes
- Frederik Rietdijk +
- Rémy Léone +
- Christian Sachs +
- Skipper Seabold
- Sebastian Skoupý +
- Alex Seewald +
- Andreas Sorge +
- Bernardo Sulzbach +
- Julian Taylor
- Louis Tiao +
- Utkarsh Upadhyay +
- Jacob Vanderplas
- Gael Varoquaux +
- Pauli Virtanen
- Fredrik Wallner +
- Stefan van der Walt
- James Webber +
- Warren Weckesser
- Raphael Wettinger +
- Josh Wilson +
- Nat Wilson +
- Peter Yin +

A total of 101 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.17.0

- #1923: problem with numpy 0's in stats.poisson.rvs (Trac #1398)
- #2138: scipy.misc.imread segfaults on 1 bit png (Trac #1613)
- #2237: distributions do not accept complex arguments (Trac #1718)
- #2282: scipy.special.hyp1f1(0.5, 1.5, -1000) fails (Trac #1763)

- #2618: poisson.pmf returns NaN if mu is 0
- #2957: hyp1f1 precision issue
- #2997: FAIL: test_qhull.TestUtilities.test_more_barycentric_transforms
- #3129: No way to set ranges for fitting parameters in Optimize functions
- #3191: interp1d should contain a fill_value_below and a fill_value_above...
- #3453: PchipInterpolator sets slopes at edges differently than Matlab's...
- #4106: ndimage._ni_support._normalize_sequence() fails with numpy.int64
- #4118: `scipy.integrate.ode.set_solout` called after `scipy.integrate.ode.set_initial_value` fails silently
- #4233: 1D scipy.interpolate.griddata using method=nearest produces nans...
- #4375: All tests fail due to bad file permissions
- #4580: scipy.ndimage.filters.convolve documenation is incorrect
- #4627: logsumexp with sign indicator - enable calculation with negative...
- #4702: logsumexp with zero scaling factor
- #4834: gammainc should return 1.0 instead of NaN for infinite x
- #4838: enh: exprel special function
- #4862: the scipy.special.boxcox function is inaccurate for denormal...
- #4887: Spherical harmonic incongruences
- #4895: some scipy ufuncs have inconsistent output dtypes?
- #4923: logm does not aggressively convert complex outputs to float
- #4932: BUG: stats: The *fit* method of the distributions silently ignores...
- #4956: Documentation error in `scipy.special.bi_zeros`
- #4957: Docstring for *pbvv_seq* is wrong
- #4967: block_diag should look at dtypes of all arguments, not only the...
- #5037: scipy.optimize.minimize error messages are printed to stdout...
- #5039: Cubic interpolation: On entry to DGESDD parameter number 12 had...
- #5163: Base case example of Hierarchical Clustering (offer)
- #5181: BUG: stats.genextreme.entropy should use the explicit formula
- #5184: Some? wheels don't express a numpy dependency
- #5197: mstats: test_kurtosis fails (ULP max is 2)
- #5260: Typo causing an error in splrep
- #5263: Default epsilon in rbf.py fails for colinear points
- #5276: Reading empty (no data) arff file fails
- #5280: 1d scipy.signal.convolve much slower than numpy.convolve
- #5326: Implementation error in scipy.interpolate.PchipInterpolator
- #5370: Test issue with test_quadpack and libm.so as a linker script

- #5426: ERROR: test_stats.test_chisquare_masked_arrays
- #5427: Automate installing correct numpy versions in numpy-vendor image
- #5430: Python3 : Numpy scalar types "not iterable"; specific instance...
- #5450: BUG: spatial.ConvexHull triggers a seg. fault when given nans.
- #5478: clarify the relation between matrix normal distribution and *multivariate_normal*
- #5539: lstsq related test failures on windows binaries from numpy-vendor
- #5560: doc: scipy.stats.burr pdf issue
- #5571: lstsq test failure after lapack_driver change
- #5577: ordqz segfault on Python 3.4 in Wine
- #5578: scipy.linalg test failures on python 3 in Wine
- #5607: Overloaded 'isnan(double&)' is ambiguous when compiling with...
- #5629: Test for lstsq randomly failed
- #5630: memory leak with scipy 0.16 spatial cKDEtree
- #5689: isnan errors compiling scipy/special/Faddeeva.cc with clang++
- #5694: fftpack test failure in test_import
- #5719: curve_fit(method!="lm") ignores initial guess

Pull requests for 0.17.0

- #3022: hyp1f1: better handling of large negative arguments
- #3107: ENH: Add ordered QZ decomposition
- #4390: ENH: Allow axis and keepdims arguments to be passed to scipy.linalg.norm.
- #4671: ENH: add axis to sparse norms
- #4796: ENH: Add cut tree function to scipy.cluster.hierarchy
- #4809: MAINT: cauchy moments are undefined
- #4821: ENH: stats: make distribution instances picklable
- #4839: ENH: Add scipy.special.exprel relative error exponential ufunc
- #4859: Logsumexp fixes - allows sign flags and b==0
- #4865: BUG: scipy.io.mmio.write: error with big indices and low precision
- #4869: add as_inexact option to _lib._util._asarray_validated
- #4884: ENH: Finite difference approximation of Jacobian matrix
- #4890: ENH: Port cKDTree query methods to C++, allow pickling on Python...
- #4892: how much doctesting is too much?
- #4896: MAINT: work around a possible numpy ufunc loop selection bug
- #4898: MAINT: A bit of pyflakes-driven cleanup.
- #4899: ENH: add 'alternative' keyword to hypothesis tests in stats
- #4903: BENCH: Benchmarks for interpolate module

- #4905: MAINT: prepend underscore to mask_to_limits; delete masked_var.
- #4906: MAINT: Benchmarks for optimize.leastsq
- #4910: WIP: Trimmed statistics functions have inconsistent API.
- #4912: MAINT: fix typo in stats tutorial. Closes gh-4911.
- #4914: DEP: deprecate `scipy.stats.ss` and `scipy.stats.square_of_sums`.
- #4924: MAINT: if the imaginary part of logm of a real matrix is small,...
- #4930: BENCH: Benchmarks for signal module
- #4941: ENH: update *find_repeats*.
- #4942: MAINT: use np.float64_t instead of np.float_t in cKDTree
- #4944: BUG: integer overflow in correlate_nd
- #4951: do not ignore invalid kwargs in distributions fit method
- #4958: Add some detail to docstrings for special functions
- #4961: ENH: stats.describe: add bias kw and empty array handling
- #4963: ENH: scipy.sparse.coo.coo_matrix.__init__: less memory needed
- #4968: DEP: deprecate `stats.f_value*` and `mstats.f_value*` functions.
- #4969: ENH: review *stats.relfreq* and *stats.cumfreq*; fixes to *stats.histogram*
- #4971: Extend github source links to line ranges
- #4972: MAINT: impove the error message in validate_runtests_log
- #4976: DEP: deprecate `scipy.stats.threshold`
- #4977: MAINT: more careful dtype treatment in block diagonal matrix...
- #4979: ENH: distributions, complex arguments
- #4984: clarify dirichlet distribution error handling
- #4992: ENH: *stats.fligner* and *stats.bartlett* empty input handling.
- #4996: DOC: fix stats.spearmanr docs
- #4997: Fix up boxcox for underflow / loss of precision
- #4998: DOC: improved documentation for *stats.ppcc_max*
- #5000: ENH: added empty input handling *scipy.moment*; doc enhancements
- #5003: ENH: improves rankdata algorithm
- #5005: scipy.stats: numerical stability improvement
- #5007: ENH: nan handling in functions that use *stats._chk_asarray*
- #5009: remove coveralls.io
- #5010: Hypergeometric distribution log survival function
- #5014: Patch to compute the volume and area of convex hulls
- #5015: DOC: Fix mistaken variable name in sawtooth
- #5016: DOC: resample example
- #5017: DEP: deprecate *stats.betai* and *stats.chisqprob*

- #5018: ENH: Add test on random inpu to volume computations
- #5026: BUG: Fix return dtype of lil_matrix.getnnz(axis=0)
- #5030: DOC: resample slow for prime output too
- #5033: MAINT: integrate, special: remove unused R1MACH and Makefile
- #5034: MAINT: signal: lift max_len_seq validation out of Cython
- #5035: DOC/MAINT: refguide / doctest drudgery
- #5041: BUG: fixing some small memory leaks detected by cppcheck
- #5044: [GSoC] ENH: New least-squares algorithms
- #5050: MAINT: C fixes, trimmed a lot of dead code from Cephes
- #5057: ENH: sparse: avoid densifying on sparse/dense elementwise mult
- #5058: TST: stats: add a sample distribution to the test loop
- #5061: ENH: spatial: faster 2D Voronoi and Convex Hull plotting
- #5065: TST: improve test coverage for *stats.mvsdist* and *stats.bayes_mvs*
- #5066: MAINT: fitpack: remove a noop
- #5067: ENH: empty and nan input handling for *stats.kstat* and *stats.kstatvar*
- #5071: DOC: optimize: Correct paper reference, add doi
- #5072: MAINT: scipy.sparse cleanup
- #5073: DOC: special: Add an example showing the relation of diric to...
- #5075: DOC: clarified parameterization of stats.lognorm
- #5076: use int, float, bool instead of np.int, np.float, np.bool
- #5078: DOC: Rename fftpack docs to README
- #5081: BUG: Correct handling of scalar 'b' in lsmr and lsqr
- #5082: loadmat variable_names: don't confuse [] and None.
- #5083: Fix integrate.fixed_quad docstring to indicate None return value
- #5086: Use solve() instead of inv() for gaussian_kde
- #5090: MAINT: stats: add explicit _sf, _isf to gengamma distribution
- #5094: ENH: scipy.interpolate.NearestNDInterpolator: cKDTree configurable
- #5098: DOC: special: fix typesetting in `*_roots quadrature` functions
- #5099: DOC: make the docstring of stats.moment raw
- #5104: DOC/ENH fixes and micro-optimizations for scipy.linalg
- #5105: enh: made l-bfgs-b parameter for the maximum number of line search...
- #5106: TST: add NIST test cases to *stats.f_oneway*
- #5110: [GSoC]: Bounded linear least squares
- #5111: MAINT: special: Cephes cleanup
- #5118: BUG: FIR path failed if len(x) < len(b) in lfilter.
- #5124: ENH: move the filliben approximation to a publicly visible function

- #5126: StatisticsCleanup: *stats.kruskal* review
- #5130: DOC: update PyPi trove classifiers. Beta -> Stable. Add license.
- #5131: DOC: differential_evolution, improve docstring for mutation and...
- #5132: MAINT: differential_evolution improve init_population_lhs comments...
- #5133: MRG: rebased mmio refactoring
- #5135: MAINT: *stats.mstats* consistency with *stats.stats*
- #5139: TST: linalg: add a smoke test for gh-5039
- #5140: EHN: Update constants.codata to CODATA 2014
- #5145: added ValueError to docstring as possible error raised
- #5146: MAINT: Improve implementation details and doc in *stats.shapiro*
- #5147: [GSoC] ENH: Upgrades to curve_fit
- #5150: Fix misleading wavelets/cwt example
- #5152: BUG: cluster.hierarchy.dendrogram: missing font size doesn't...
- #5153: add keywords to control the summation in discrete distributions...
- #5156: DOC: added comments on algorithms used in Legendre function
- #5158: ENH: optimize: add the Hungarian algorithm
- #5162: FIX: Remove lena
- #5164: MAINT: fix cluster.hierarchy.dendrogram issues and docs
- #5166: MAINT: changed *stats.pointbiserialr* to delegate to *stats.pearsonr*
- #5167: ENH: add nan_policy to *stats.kendalltau*.
- #5168: TST: added nist test case (Norris) to *stats.linregress*.
- #5169: update lpmv docstring
- #5171: Clarify metric parameter in linkage docstring
- #5172: ENH: add mode keyword to signal.spectrogram
- #5177: DOC: graphical example for KDTree.query_ball_point
- #5179: MAINT: stats: tweak the formula for ncx2.pdf
- #5188: MAINT: linalg: A bit of clean up.
- #5189: BUG: stats: Use the explicit formula in stats.genextreme.entropy
- #5193: BUG: fix uninitialized use in lartg
- #5194: BUG: properly return error to fortran from ode_jacobian_function
- #5198: TST: Fix TestCtypesQuad failure on Python 3.5 for Windows
- #5201: allow extrapolation in interp1d
- #5209: MAINT: Change complex parameter to boolean in Y_()
- #5213: BUG: sparse: fix logical comparison dtype conflicts
- #5216: BUG: sparse: fixing unbound local error
- #5218: DOC and BUG: Bessel function docstring improvements, fix array_like,...

- #5222: MAINT: sparse: fix COO ctor
- #5224: DOC: optimize: type of OptimizeResult.hess_inv varies
- #5228: ENH: Add maskandscale support to netcdf; based on pupynere and...
- #5229: DOC: sparse.linalg.svds doc typo fixed
- #5234: MAINT: sparse: simplify COO ctor
- #5235: MAINT: sparse: warn on todia() with many diagonals
- #5236: MAINT: ndimage: simplify thread handling/recursion + constness
- #5239: BUG: integrate: Fixed issue 4118
- #5241: qr_insert fixes, closes #5149
- #5246: Doctest tutorial files
- #5247: DOC: optimize: typo/import fix in linear_sum_assignment
- #5248: remove inspect.getargspec and test python 3.5 on Travis CI
- #5250: BUG: Fix sparse multiply by single-element zero
- #5261: Fix bug causing a TypeError in splrep when a runtime warning...
- #5262: Follow up to 4489 (Addition LAPACK routines in linalg.lstsq)
- #5264: ignore zero-length edges for default epsilon
- #5269: DOC: Typos and spell-checking
- #5272: MAINT: signal: Convert array syntax to memoryviews
- #5273: DOC: raw strings for docstrings with math
- #5274: MAINT: sparse: update cython code for MST
- #5278: BUG: io: Stop guessing the data delimiter in ARFF files.
- #5289: BUG: misc: Fix the Pillow work-around for 1-bit images.
- #5291: ENH: call np.correlate for 1d in scipy.signal.correlate
- #5294: DOC: special: Remove a potentially misleading example from the...
- #5295: Simplify replacement of fftpack by pyfftw
- #5296: ENH: Add matrix normal distribution to stats
- #5297: Fixed leaf_rotation and leaf_font_size in Python 3
- #5303: MAINT: stats: rewrite find_repeats
- #5307: MAINT: stats: remove unused Fortran routine
- #5313: BUG: sparse: fix diags for nonsquare matrices
- #5315: MAINT: special: Cephes cleanup
- #5316: fix input check for sparse.linalg.svds
- #5319: MAINT: Cython code maintenance
- #5328: BUG: Fix place_poles return values
- #5329: avoid a spurious divide-by-zero in Student t stats
- #5334: MAINT: integrate: miscellaneous cleanup

- #5340: MAINT: Printing Error Msg to STDERR and Removing iterate.dat
- #5347: ENH: add Py3.5-style matmul operator (e.g. A @ B) to sparse linear...
- #5350: FIX error, when reading 32-bit float wav files
- #5351: refactor the PCHIP interpolant's algorithm
- #5354: MAINT: construct csr and csc matrices from integer lists
- #5359: add a fast path to interp1d
- #5364: Add two fill_values to interp1d.
- #5365: ABCD docstrings
- #5366: Fixed typo in the documentation for scipy.signal.cwt() per #5290.
- #5367: DOC updated scipy.spatial.Delaunay example
- #5368: ENH: Do not create a throwaway class at every function call
- #5372: DOC: spectral: fix reference formatting
- #5375: PEP8 amendments to ffpack_basic.py
- #5377: BUG: integrate: builtin name no longer shadowed
- #5381: PEP8ified fftpack_pseudo_diffs.py
- #5385: BLD: fix Bento build for changes to optimize and spatial
- #5386: STY: PEP8 amendments to interpolate.py
- #5387: DEP: deprecate stats.histogram
- #5388: REL: add "make upload" command to doc/Makefile.
- #5389: DOC: updated origin param of scipy.ndimage.filters.convolve
- #5395: BUG: special: fix a number of edge cases related to $x = np.inf$.
- #5398: MAINT: stats: avoid spurious warnings in lognorm.pdf(0, s)
- #5407: ENH: stats: Handle mu=0 in stats.poisson
- #5409: Fix the behavior of discrete distributions at the right-hand...
- #5412: TST: stats: skip a test to avoid a spurious log(0) warning
- #5413: BUG: linalg: work around LAPACK single-precision lwork computation...
- #5414: MAINT: stats: move creation of namedtuples outside of function...
- #5415: DOC: fix up sections in ToC in the pdf reference guide
- #5416: TST: fix issue with a ctypes test for integrate on Fedora.
- #5418: DOC: fix bugs in signal.TransferFunction docstring. Closes gh-5287.
- #5419: MAINT: sparse: fix usage of NotImplementedError
- #5420: Raise proper error if maxiter < 1
- #5422: DOC: changed documentation of brent to be consistent with bracket
- #5444: BUG: gaussian_filter, BPoly.from_derivatives fail on numpy int...
- #5445: MAINT: stats: fix incorrect deprecation warnings and test noise
- #5446: DOC: add note about PyFFTW in fftpack tutorial.

- #5459: DOC: integrate: Some improvements to the differential equation...
- #5465: BUG: Relax mstats kurtosis test tolerance by a few ulp
- #5471: ConvexHull should raise ValueError for NaNs.
- #5473: MAINT: update decorators.py module to version 4.0.5
- #5476: BUG: imsave searches for wrong channel axis if image has 3 or...
- #5477: BLD: add numpy to setup/install_requires for OS X wheels
- #5479: ENH: return Jacobian/Hessian from BasinHopping
- #5484: BUG: fix ttest zero division handling
- #5486: Fix crash on kmeans2
- #5491: MAINT: Expose parallel build option to runtests.py
- #5494: Sort OptimizeResult.__repr__ by key
- #5496: DOC: update the author name mapping
- #5497: Enhancement to binned_statistic: option to unraveled returned...
- #5498: BUG: sparse: fix a bug in sparsetools input dtype resolution
- #5500: DOC: detect unprintable characters in docstrings
- #5505: BUG: misc: Ensure fromimage converts mode 'P' to 'RGB' or 'RGBA'.
- #5514: DOC: further update the release notes
- #5515: ENH: optionally disable fixed-point acceleration
- #5517: DOC: Improvements and additions to the matrix_normal doc
- #5518: Remove wrappers for LAPACK deprecated routines
- #5521: TST: skip a linalg.orth memory test on 32-bit platforms.
- #5523: DOC: change a few floats to integers in docstring examples
- #5524: DOC: more updates to 0.17.0 release notes.
- #5525: Fix to minor typo in documentation for scipy.integrate.ode
- #5527: TST: bump arccosh tolerance to allow for inaccurate numpy or...
- #5535: DOC: signal: minor clarification to docstring of TransferFunction.
- #5538: DOC: signal: fix find_peaks_cwt documentation
- #5545: MAINT: Fix typo in linalg/basic.py
- #5547: TST: mark TestEig.test_singular as knownfail in master.
- #5550: MAINT: work around lstsq driver selection issue
- #5556: BUG: Fixed broken dogbox trust-region radius update
- #5561: BUG: eliminate warnings, exception (on Win) in test_maskandscale;...
- #5567: TST: a few cleanups in the test suite; run_module_suite and clearer...
- #5568: MAINT: simplify poisson's _argcheck
- #5569: TST: bump GMean test tolerance to make it pass on Wine
- #5572: TST: lstsq: bump test tolerance for TravisCI

- #5573: TST: remove use of np.fromfile from cluster.vq tests
- #5576: Lapack deprecations
- #5579: TST: skip tests of linalg.norm axis keyword on numpy <= 1.7.x
- #5582: Clarify language of survival function documentation
- #5583: MAINT: stats/tests: A bit of clean up.
- #5588: DOC: stats: Add a note that stats.burr is the Type III Burr distribution.
- #5595: TST: fix test_lamch failures on Python 3
- #5600: MAINT: Ignore spatial/ckdtree.cxx and .h
- #5602: Explicitly numbered replacement fields for maintainability
- #5605: MAINT: collection of small fixes to test suite
- #5614: Minor doc change.
- #5624: FIX: Fix interpolate
- #5625: BUG: msvc9 binaries crash when indexing std::vector of size 0
- #5635: BUG: misspelled __dealloc__ in cKDTree.
- #5642: STY: minor fixup of formatting of 0.17.0 release notes.
- #5643: BLD: fix a build issue in special/Faddeeva.cc with isnan.
- #5661: TST: linalg tests used stdlib random instead of numpy.random.
- #5682: backports for 0.17.0
- #5696: Minor improvements to least_squares' docstring.
- #5697: BLD: fix for isnan/isinf issues in special/Faddeeva.cc
- #5720: TST: fix for file opening error in fftpack test_import.py
- #5722: BUG: Make curve_fit respect an initial guess with bounds
- #5726: Backports for v0.17.0rc2
- #5727: API: Changes to least_squares API

1.7 SciPy 0.16.1 Release Notes

SciPy 0.16.1 is a bug-fix release with no new features compared to 0.16.0.

1.7.1 Issues closed for 0.16.1

- #5077: cKDTree not indexing properly for arrays with too many elements
- #5127: Regression in 0.16.0: solve_banded errors out in patsy test suite
- #5149: linalg tests apparently cause python to crash with numpy 1.10.0b1
- #5154: 0.16.0 fails to build on OS X; can't find Python.h
- #5173: failing stats.histogram test with numpy 1.10
- #5191: Scipy 0.16.x - TypeError: _asarray_validated() got an unexpected...

- #5195: tarballs missing documentation source
- #5363: FAIL: test_orthogonal.test_j_roots, test_orthogonal.test_js_roots

1.7.2 Pull requests for 0.16.1

- #5088: BUG: fix logic error in cKDTree.sparse_distance_matrix
- #5089: BUG: Don't overwrite b in lfilter's FIR path
- #5128: BUG: solve_banded failed when solving 1x1 systems
- #5155: BLD: fix missing Python include for Homebrew builds.
- #5192: BUG: backport as_inexact kwarg to _asarray_validated
- #5203: BUG: fix uninitialized use in lartg 0.16 backport
- #5204: BUG: properly return error to fortran from ode_jacobian_function...
- #5207: TST: Fix TestCtypesQuad failure on Python 3.5 for Windows
- #5352: TST: sparse: silence warnings about boolean indexing
- #5355: MAINT: backports for 0.16.1 release
- #5356: REL: update Paver file to ensure sdist contents are OK for releases.
- #5382: 0.16.x backport: MAINT: work around a possible numpy ufunc loop...
- #5393: TST:special: bump tolerance levels for test_j_roots and test_js_roots
- #5417: MAINT: stats: move namedtuple creating outside function calls.

1.8 SciPy 0.16.0 Release Notes

Contents

- *SciPy 0.16.0 Release Notes*
  - *New features*
    - *Benchmark suite*
    - `scipy.linalg` *improvements*
    - `scipy.signal` *improvements*
    - `scipy.sparse` *improvements*
    - `scipy.spatial` *improvements*
    - `scipy.stats` *improvements*
    - `scipy.optimize` *improvements*
  - *Deprecated features*
  - *Backwards incompatible changes*
  - *Other changes*
  - *Authors*

* *Issues closed for 0.16.0*
* *Pull requests for 0.16.0*

SciPy 0.16.0 is the culmination of 7 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.16.x branch, and on adding new features on the master branch.

This release requires Python 2.6, 2.7 or 3.2-3.4 and NumPy 1.6.2 or greater.

Highlights of this release include:

- A Cython API for BLAS/LAPACK in `scipy.linalg`
- A new benchmark suite. It's now straightforward to add new benchmarks, and they're routinely included with performance enhancement PRs.
- Support for the second order sections (SOS) format in `scipy.signal`.

1.8.1 New features

Benchmark suite

The benchmark suite has switched to using Airspeed Velocity for benchmarking. You can run the suite locally via `python runtests.py --bench`. For more details, see `benchmarks/README.rst`.

`scipy.linalg` improvements

A full set of Cython wrappers for BLAS and LAPACK has been added in the modules `scipy.linalg.cython_blas` and `scipy.linalg.cython_lapack`. In Cython, these wrappers can now be cimported from their corresponding modules and used without linking directly against BLAS or LAPACK.

The functions `scipy.linalg.qr_delete`, `scipy.linalg.qr_insert` and `scipy.linalg.qr_update` for updating QR decompositions were added.

The function `scipy.linalg.solve_circulant` solves a linear system with a circulant coefficient matrix.

The function `scipy.linalg.invpascal` computes the inverse of a Pascal matrix.

The function `scipy.linalg.solve_toeplitz`, a Levinson-Durbin Toeplitz solver, was added.

Added wrapper for potentially useful LAPACK function `*lasd4`. It computes the square root of the i-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. See its LAPACK documentation and unit tests for it to get more info.

Added two extra wrappers for LAPACK least-square solvers. Namely, they are `*gelsd` and `*gelsy`.

Wrappers for the LAPACK `*lange` functions, which calculate various matrix norms, were added.

Wrappers for `*gtsv` and `*ptsv`, which solve `A*X = B` for tri-diagonal matrix `A`, were added.

`scipy.signal` improvements

Support for second order sections (SOS) as a format for IIR filters was added. The new functions are:

- `scipy.signal.sosfilt`

- *scipy.signal.sosfilt_zi*,
- *scipy.signal.sos2tf*
- *scipy.signal.sos2zpk*
- *scipy.signal.tf2sos*
- *scipy.signal.zpk2sos*.

Additionally, the filter design functions *iirdesign*, *iirfilter*, *butter*, *cheby1*, *cheby2*, *ellip*, and *bessel* can return the filter in the SOS format.

The function *scipy.signal.place_poles*, which provides two methods to place poles for linear systems, was added.

The option to use Gustafsson's method for choosing the initial conditions of the forward and backward passes was added to *scipy.signal.filtfilt*.

New classes `TransferFunction`, `StateSpace` and `ZerosPolesGain` were added. These classes are now returned when instantiating *scipy.signal.lti*. Conversion between those classes can be done explicitly now.

An exponential (Poisson) window was added as *scipy.signal.exponential*, and a Tukey window was added as *scipy.signal.tukey*.

The function for computing digital filter group delay was added as *scipy.signal.group_delay*.

The functionality for spectral analysis and spectral density estimation has been significantly improved: *scipy.signal.welch* became ~8x faster and the functions *scipy.signal.spectrogram*, *scipy.signal.coherence* and *scipy.signal.csd* (cross-spectral density) were added.

scipy.signal.lsim was rewritten - all known issues are fixed, so this function can now be used instead of `lsim2`; `lsim` is orders of magnitude faster than `lsim2` in most cases.

`scipy.sparse` improvements

The function *scipy.sparse.norm*, which computes sparse matrix norms, was added.

The function *scipy.sparse.random*, which allows to draw random variates from an arbitrary distribution, was added.

`scipy.spatial` improvements

scipy.spatial.cKDTree has seen a major rewrite, which improved the performance of the `query` method significantly, added support for parallel queries, pickling, and options that affect the tree layout. See pull request 4374 for more details.

The function *scipy.spatial.procrustes* for Procrustes analysis (statistical shape analysis) was added.

`scipy.stats` improvements

The Wishart distribution and its inverse have been added, as *scipy.stats.wishart* and *scipy.stats.invwishart*.

The Exponentially Modified Normal distribution has been added as *scipy.stats.exponnorm*.

The Generalized Normal distribution has been added as *scipy.stats.gennorm*.

All distributions now contain a `random_state` property and allow specifying a specific `numpy.random.RandomState` random number generator when generating random variates.

Many statistical tests and other *scipy.stats* functions that have multiple return values now return namedtuples. See pull request 4709 for details.

scipy.optimize improvements

A new derivative-free method DF-SANE has been added to the nonlinear equation system solving function *scipy.optimize.root*.

1.8.2 Deprecated features

scipy.stats.pdf_fromgamma is deprecated. This function was undocumented, untested and rarely used. Statsmodels provides equivalent functionality with statsmodels.distributions.ExpandedNormal.

scipy.stats.fastsort is deprecated. This function is unnecessary, numpy.argsort can be used instead.

scipy.stats.signaltonoise and scipy.stats.mstats.signaltonoise are deprecated. These functions did not belong in scipy.stats and are rarely used. See issue #609 for details.

scipy.stats.histogram2 is deprecated. This function is unnecessary, numpy.histogram2d can be used instead.

1.8.3 Backwards incompatible changes

The deprecated global optimizer scipy.optimize.anneal was removed.

The following deprecated modules have been removed: scipy.lib.blas, scipy.lib.lapack, scipy.linalg.cblas, scipy.linalg.fblas, scipy.linalg.clapack, scipy.linalg.flapack. They had been deprecated since Scipy 0.12.0, the functionality should be accessed as *scipy.linalg.blas* and *scipy.linalg.lapack*.

The deprecated function scipy.special.all_mat has been removed.

The deprecated functions fprob, ksprob, zprob, randwcdf and randwppf have been removed from *scipy.stats*.

1.8.4 Other changes

The version numbering for development builds has been updated to comply with PEP 440.

Building with python setup.py develop is now supported.

1.8.5 Authors

- @axiru +
- @endolith
- Elliott Sales de Andrade +
- Anne Archibald
- Yoshiki Vázquez Baeza +
- Sylvain Bellemare
- Felix Berkenkamp +

- Raoul Bourquin +
- Matthew Brett
- Per Brodtkorb
- Christian Brueffer
- Lars Buitinck
- Evgeni Burovski
- Steven Byrnes
- CJ Carey
- George Castillo +
- Alex Conley +
- Liam Damewood +
- Rupak Das +
- Abraham Escalante +
- Matthias Feurer +
- Eric Firing +
- Clark Fitzgerald
- Chad Fulton
- André Gaul
- Andreea Georgescu +
- Christoph Gohlke
- Andrey Golovizin +
- Ralf Gommers
- J.J. Green +
- Alex Griffing
- Alexander Grigorievskiy +
- Hans Moritz Gunther +
- Jonas Hahnfeld +
- Charles Harris
- Ian Henriksen
- Andreas Hilboll
- Åsmund Hjulstad +
- Jan Schlüter +
- Janko Slavič +
- Daniel Jensen +
- Johannes Ballé +
- Terry Jones +

- Amato Kasahara +
- Eric Larson
- Denis Laxalde
- Antony Lee
- Gregory R. Lee
- Perry Lee +
- Loïc Estève
- Martin Manns +
- Eric Martin +
- Matěj Kocián +
- Andreas Mayer +
- Nikolay Mayorov +
- Robert McGibbon +
- Sturla Molden
- Nicola Montecchio +
- Eric Moore
- Jamie Morton +
- Nikolas Moya +
- Maniteja Nandana +
- Andrew Nelson
- Joel Nothman
- Aldrian Obaja
- Regina Ongowarsito +
- Paul Ortyl +
- Pedro López-Adeva Fernández-Layos +
- Stefan Peterson +
- Irvin Probst +
- Eric Quintero +
- John David Reaver +
- Juha Remes +
- Thomas Robitaille
- Clancy Rowley +
- Tobias Schmidt +
- Skipper Seabold
- Aman Singh +
- Eric Soroos

- Valentine Svensson +
- Julian Taylor
- Aman Thakral +
- Helmut Toplitzer +
- Fukumu Tsutsumi +
- Anastasiia Tsyplia +
- Jacob Vanderplas
- Pauli Virtanen
- Matteo Visconti +
- Warren Weckesser
- Florian Wilhelm +
- Nathan Woods
- Haochen Wu +
- Daan Wynen +

A total of 93 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed for 0.16.0

- #1063: Implement a whishart distribution (Trac #536)
- #1885: Rbf: floating point warnings - possible bug (Trac #1360)
- #2020: Rbf default epsilon too large (Trac #1495)
- #2325: extending distributions, hypergeom, to degenerate cases (Trac...
- #3502: [ENH] linalg.hessenberg should use ORGHR for calc_q=True
- #3603: Passing array as window into signal.resample() fails
- #3675: Intermittent failures for signal.slepian on Windows
- #3742: Pchipinterpolator inconvenient as ppoly
- #3786: add procrustes?
- #3798: scipy.io.savemat fails for empty dicts
- #3975: Use RandomState in scipy.stats
- #4022: savemat incorrectly saves logical arrays
- #4028: scipy.stats.geom.logpmf(1,1) returns nan. The correct value is...
- #4030: simplify scipy.stats.betaprime.cdf
- #4031: improve accuracy of scipy.stats.gompertz distribution for small...
- #4033: improve accuracy of scipy.stats.lomax distribution for small...
- #4034: improve accuracy of scipy.stats.rayleigh distribution for large...
- #4035: improve accuracy of scipy.stats.truncexpon distribution for small...

- #4081: Error when reading matlab file: buffer is too small for requested...
- #4100: Why does qr(a, lwork=0) not fail?
- #4134: scipy.stats: rv_frozen has no expect() method
- #4204: Please add docstring to scipy.optimize.RootResults
- #4206: Wrap LAPACK tridiagonal solve routine *gtsv*
- #4208: Empty sparse matrices written to MAT file cannot be read by MATLAB
- #4217: use a TravisCI configuration with numpy built with NPY_RELAXED_STRIDES_CHECKING=1
- #4282: integrate.odeint raises an exception when full_output=1 and the...
- #4301: scipy and numpy version names do not follow pep 440
- #4355: PPoly.antiderivative() produces incorrect output
- #4391: spsolve becomes extremely slow with large b matrix
- #4393: Documentation glitsch in sparse.linalg.spilu
- #4408: Vector-valued constraints in minimize() et al
- #4412: Documentation of scipy.signal.cwt error
- #4428: dok.__setitem__ problem with negative indices
- #4434: Incomplete documentation for sparse.linalg.spsolve
- #4438: linprog() documentation example wrong
- #4445: Typo in scipy.special.expit doc
- #4467: Documentation Error in scipy.optimize options for TNC
- #4492: solve_toeplitz benchmark is bitrotting already
- #4506: lobpcg/sparse performance regression Jun 2014?
- #4520: g77_abi_wrappers needed on Linux for MKL as well
- #4521: Broken check in uses_mkl for newer versions of the library
- #4523: rbf with gaussian kernel seems to produce more noise than original...
- #4526: error in site documentation for poisson.pmf() method
- #4527: KDTree example doesn't work in Python 3
- #4550: `scipy.stats.mode` - UnboundLocalError on empty sequence
- #4554: filter out convergence warnings in optimization tests
- #4565: odeint messages
- #4569: remez: "ValueError: Failure to converge after 25 iterations....
- #4582: DOC: optimize: _minimize_scalar_brent does not have a disp option
- #4585: DOC: Erroneous latex-related characters in tutorial.
- #4590: sparse.linalg.svds should throw an exception if which not in...
- #4594: scipy.optimize.linprog IndexError when a callback is providen
- #4596: scipy.linalg.block_diag misbehavior with empty array inputs (v0.13.3)
- #4599: scipy.integrate.nquad should call _OptFunc when called with only...

- #4612: Crash in signal.lfilter on nd input with wrong shaped zi
- #4613: scipy.io.readsav error on reading sav file
- #4673: scipy.interpolate.RectBivariateSpline construction locks PyQt...
- #4681: Broadcasting in signal.lfilter still not quite right.
- #4705: kmeans k_or_guess parameter error if guess is not square array
- #4719: Build failure on 14.04.2
- #4724: GenGamma _munp function fails due to overflow
- #4726: FAIL: test_cobyla.test_vector_constraints
- #4734: Failing tests in stats with numpy master.
- #4736: qr_update bug or incompatibility with numpy 1.10?
- #4746: linprog returns solution violating equality constraint
- #4757: optimize.leastsq docstring mismatch
- #4774: Update contributor list for v0.16
- #4779: circmean and others do not appear in the documentation
- #4788: problems with scipy sparse linalg isolve iterative.py when complex
- #4791: BUG: scipy.spatial: incremental Voronoi doesn't increase size...

Pull requests for 0.16.0

- #3116: sparse: enhancements for DIA format
- #3157: ENH: linalg: add the function 'solve_circulant' for solving a...
- #3442: ENH: signal: Add Gustafsson's method as an option for the filtfilt...
- #3679: WIP: fix sporadic slepian failures
- #3680: Some cleanups in stats
- #3717: ENH: Add second-order sections filtering
- #3741: Dltisys changes
- #3956: add note to scipy.signal.resample about prime sample numbers
- #3980: Add check_finite flag to UnivariateSpline
- #3996: MAINT: stricter linalg argument checking
- #4001: BUG: numerical precision in dirichlet
- #4012: ENH: linalg: Add a function to compute the inverse of a Pascal...
- #4021: ENH: Cython api for lapack and blas
- #4089: Fixes for various PEP8 issues.
- #4116: MAINT: fitpack: trim down compiler warnings (unused labels, variables)
- #4129: ENH: stats: add a random_state property to distributions
- #4135: ENH: Add Wishart and inverse Wishart distributions
- #4195: improve the interpolate docs

- #4200: ENH: Add t-test from descriptive stats function.
- #4202: Dendrogram threshold color
- #4205: BLD: fix a number of Bento build warnings.
- #4211: add an ufunc for the inverse Box-Cox transfrom
- #4212: MRG:fix for gh-4208
- #4213: ENH: specific warning if matlab file is empty
- #4215: Issue #4209: splprep documentation updated to reflect dimensional...
- #4219: DOC: silence several Sphinx warnings when building the docs
- #4223: MAINT: remove two redundant lines of code
- #4226: try forcing the numpy rebuild with relaxed strides
- #4228: BLD: some updates to Bento config files and docs. Closes gh-3978.
- #4232: wrong references in the docs
- #4242: DOC: change example sample spacing
- #4245: Arff fixes
- #4246: MAINT: C fixes
- #4247: MAINT: remove some unused code
- #4249: Add routines for updating QR decompositions
- #4250: MAINT: Some pyflakes-driven cleanup in linalg and sparse
- #4252: MAINT trim away >10 kLOC of generated C code
- #4253: TST: stop shadowing ellip* tests vs boost data
- #4254: MAINT: special: use NPY_PI, not M_PI
- #4255: DOC: INSTALL: use Py3-compatible print syntax, and don't mention...
- #4256: ENH: spatial: reimplement cdist_cosine using np.dot
- #4258: BUG: io.arff #4429 #2088
- #4261: MAINT: signal: PEP8 and related style clean up.
- #4262: BUG: newton_krylov() was ignoring norm_tol argument, closes #4259
- #4263: MAINT: clean up test noise and optimize tests for docstrings...
- #4266: MAINT: io: Give an informative error when attempting to read...
- #4268: MAINT: fftpack benchmark integer division vs true division
- #4269: MAINT: avoid shadowing the eigvals function
- #4272: BUG: sparse: Fix bench_sparse.py
- #4276: DOC: remove confusing parts of the documentation related to writing...
- #4281: Sparse matrix multiplication: only convert array if needed (with...
- #4284: BUG: integrate: odeint crashed when the integration time was...
- #4286: MRG: fix matlab output type of logical array
- #4287: DEP: deprecate stats.pdf_fromgamma. Closes gh-699.

- #4291: DOC: linalg: fix layout in cholesky_banded docstring
- #4292: BUG: allow empty dict as proxy for empty struct
- #4293: MAINT: != -> not_equal in hamming distance implementation
- #4295: Pole placement
- #4296: MAINT: some cleanups in tests of several modules
- #4302: ENH: Solve toeplitz linear systems
- #4306: Add benchmark for conjugate gradient solver.
- #4307: BLD: PEP 440
- #4310: BUG: make stats.geom.logpmf(1,1) return 0.0 instead of nan
- #4311: TST: restore a test that uses slogdet now that we have dropped...
- #4313: Some minor fixes for stats.wishart addition.
- #4315: MAINT: drop numpy 1.5 compatibility code in sparse matrix tests
- #4318: ENH: Add random_state to multivariate distributions
- #4319: MAINT: fix hamming distance regression for exotic arrays, with...
- #4320: TST: a few changes like self.assertTrue(x == y, message) -> assert_equal(x,...
- #4321: TST: more changes like self.assertTrue(x == y, message) -> assert_equal(x,...
- #4322: TST: in test_signaltools, changes like self.assertTrue(x == y,...
- #4323: MAINT: clean up benchmarks so they can all be run as single files.
- #4324: Add more detailed committer guidelines, update MAINTAINERS.txt
- #4326: TST: use numpy.testing in test_hierarchy.py
- #4329: MAINT: stats: rename check_random_state test function
- #4330: Update distance tests
- #4333: MAINT: import comb, factorial from scipy.special, not scipy.misc
- #4338: TST: more conversions from nose to numpy.testing
- #4339: MAINT: remove the deprecated all_mat function from special_matrices.py
- #4340: add several features to frozen distributions
- #4344: BUG: Fix/test invalid lwork param in qr
- #4345: Fix test noise visible with Python 3.x
- #4347: Remove deprecated blas/lapack imports, rename lib to _lib
- #4349: DOC: add a nontrivial example to stats.binned_statistic.
- #4350: MAINT: remove optimize.anneal for 0.16.0 (was deprecated in 0.14.0).
- #4351: MAINT: fix usage of deprecated Numpy C API in optimize...
- #4352: MAINT: fix a number of special test failures
- #4353: implement cdf for betaprime distribution
- #4357: BUG: piecewise polynomial antiderivative
- #4358: BUG: integrate: fix handling of banded Jacobians in odeint, plus...

- #4359: MAINT: remove a code path taken for Python version < 2.5
- #4360: MAINT: stats.mstats: Remove some unused variables (thanks, pyflakes).
- #4362: Removed erroneous reference to smoothing parameter #4072
- #4363: MAINT: interpolate: clean up in fitpack.py
- #4364: MAINT: lib: don't export "partial" from decorator
- #4365: svdvals now returns a length-0 sequence of singular values given...
- #4367: DOC: slightly improve TeX rendering of wishart/invwishart docstring
- #4373: ENH: wrap gtsv and ptsv for solve_banded and solveh_banded.
- #4374: ENH: Enhancements to spatial.cKDTree
- #4376: BF: fix reading off-spec matlab logical sparse
- #4377: MAINT: integrate: Clean up some Fortran test code.
- #4378: MAINT: fix usage of deprecated Numpy C API in signal
- #4380: MAINT: scipy.optimize, removing further anneal references
- #4381: ENH: Make DCT and DST accept int and complex types like fft
- #4392: ENH: optimize: add DF-SANE nonlinear derivative-free solver
- #4394: Make reordering algorithms 64-bit clean
- #4396: BUG: bundle cblas.h in Accelerate ABI wrappers to enable compilation...
- #4398: FIX pdist bug where wminkowski's w.dtype != double
- #4402: BUG: fix stat.hypergeom argcheck
- #4404: MAINT: Fill in the full symmetric squareform in the C loop
- #4405: BUG: avoid X += X.T (refs #4401)
- #4407: improved accuracy of gompertz distribution for small x
- #4414: DOC:fix error in scipy.signal.cwt documentation.
- #4415: ENH: Improve accuracy of lomax for small x.
- #4416: DOC: correct a parameter name in docstring of SuperLU.solve....
- #4419: Restore scipy.linalg.calc_lwork also in master
- #4420: fix a performance issue with a sparse solver
- #4423: ENH: improve rayleigh accuracy for large x.
- #4424: BUG: optimize.minimize: fix overflow issue with integer x0 input.
- #4425: ENH: Improve accuracy of truncexpon for small x
- #4426: ENH: improve rayleigh accuracy for large x.
- #4427: MAINT: optimize: cleanup of TNC code
- #4429: BLD: fix build failure with numpy 1.7.x and 1.8.x.
- #4430: BUG: fix a sparse.dok_matrix set/get copy-paste bug
- #4433: Update _minimize.py
- #4435: ENH: release GIL around batch distance computations

- #4436: Fixed incomplete documentation for spsolve
- #4439: MAINT: integrate: Some clean up in the tests.
- #4440: Fast permutation t-test
- #4442: DOC: optimize: fix wrong result in docstring
- #4447: DOC: signal: Some additional documentation to go along with the...
- #4448: DOC: tweak the docstring of lapack.linalg module
- #4449: fix a typo in the expit docstring
- #4451: ENH: vectorize distance loops with gcc
- #4456: MAINT: don't fail large data tests on MemoryError
- #4461: CI: use travis_retry to deal with network timeouts
- #4462: DOC: rationalize minimize() et al. documentation
- #4470: MAINT: sparse: inherit dok_matrix.toarray from spmatrix
- #4473: BUG: signal: Fix validation of the zi shape in sosfilt.
- #4475: BLD: setup.py: update min numpy version and support "setup.py...
- #4481: ENH: add a new linalg special matrix: the Helmert matrix
- #4485: MRG: some changes to allow reading bad mat files
- #4490: [ENH] linalg.hessenberg: use orghr - rebase
- #4491: ENH: linalg: Adding wrapper for potentially useful LAPACK function...
- #4493: BENCH: the solve_toeplitz benchmark used outdated syntax and...
- #4494: MAINT: stats: remove duplicated code
- #4496: References added for watershed_ift algorithm
- #4499: DOC: reshuffle stats distributions documentation
- #4501: Replace benchmark suite with airspeed velocity
- #4502: SLSQP should strictly satisfy bound constraints
- #4503: DOC: forward port 0.15.x release notes and update author name...
- #4504: ENH: option to avoid computing possibly unused svd matrix
- #4505: Rebase of PR 3303 (sparse matrix norms)
- #4507: MAINT: fix lobpcg performance regression
- #4509: DOC: sparse: replace dead link
- #4511: Fixed differential evolution bug
- #4512: Change to fully PEP440 compliant dev version numbers (always...
- #4525: made tiny style corrections (pep8)
- #4533: Add exponentially modified gaussian distribution (scipy.stats.expongauss)
- #4534: MAINT: benchmarks: make benchmark suite importable on all scipy...
- #4535: BUG: Changed zip() to list(zip()) so that it could work in Python...
- #4536: Follow up to pr 4348 (exponential window)

- #4540: ENH: spatial: Add procrustes analysis
- #4541: Bench fixes
- #4542: TST: NumpyVersion dev -> dev0
- #4543: BUG: Overflow in savgol_coeffs
- #4544: pep8 fixes for stats
- #4546: MAINT: use reduction axis arguments in one-norm estimation
- #4549: ENH : Added group_delay to scipy.signal
- #4553: ENH: Significantly faster moment function
- #4556: DOC: document the changes of the sparse.linalg.svds (optional...
- #4559: DOC: stats: describe loc and scale parameters in the docstring...
- #4563: ENH: rewrite of stats.ppcc_plot
- #4564: Be more (or less) forgiving when user passes +-inf instead of...
- #4566: DEP: remove a bunch of deprecated function from scipy.stats,...
- #4570: MNT: Suppress LineSearchWarning's in scipy.optimize tests
- #4572: ENH: Extract inverse hessian information from L-BFGS-B
- #4576: ENH: Split signal.lti into subclasses, part of #2912
- #4578: MNT: Reconcile docstrings and function signatures
- #4581: Fix build with Intel MKL on Linux
- #4583: DOC: optimize: remove references to unused disp kwarg
- #4584: ENH: scipy.signal - Tukey window
- #4587: Hermite asymptotic
- #4593: DOC - add example to RegularGridInterpolator
- #4595: DOC: Fix erroneous latex characters in tutorial/optimize.
- #4600: Add return codes to optimize.tnc docs
- #4603: ENH: Wrap LAPACK `*lange` functions for matrix norms
- #4604: scipy.stats: generalized normal distribution
- #4609: MAINT: interpolate: fix a few inconsistencies between docstrings...
- #4610: MAINT: make runtest.py –bench-compare use asv continuous and...
- #4611: DOC: stats: explain rice scaling; add a note to the tutorial...
- #4614: BUG: lfilter, the size of zi was not checked correctly for nd...
- #4617: MAINT: integrate: Clean the C code behind odeint.
- #4618: FIX: Raise error when window length != data length
- #4619: Issue #4550: *scipy.stats.mode* - UnboundLocalError on empty...
- #4620: Fixed a problem (#4590) with svds accepting wrong eigenvalue...
- #4621: Speed up special.ai_zeros/bi_zeros by 10x
- #4623: MAINT: some tweaks to spatial.procrustes (private file, html...

- #4628: Speed up signal.lfilter and add a convolution path for FIR filters
- #4629: Bug: integrate.nquad; resolve issue #4599
- #4631: MAINT: integrate: Remove unused variables in a Fortran test function.
- #4633: MAINT: Fix convergence message for remez
- #4635: PEP8: indentation (so that pep8 bot does not complain)
- #4637: MAINT: generalize a sign function to do the right thing for complex...
- #4639: Amended typo in apple_sgemv_fix.c
- #4642: MAINT: use lapack for scipy.linalg.norm
- #4643: RBF default epsilon too large 2020
- #4646: Added atleast_1d around poly in invres and invresz
- #4647: fix doc pdf build
- #4648: BUG: Fixes #4408: Vector-valued constraints in minimize() et...
- #4649: Vonmisesfix
- #4650: Signal example clean up in Tukey and place_poles
- #4652: DOC: Fix the error in convolve for same mode
- #4653: improve erf performance
- #4655: DEP: deprecate scipy.stats.histogram2 in favour of np.histogram2d
- #4656: DEP: deprecate scipy.stats.signaltonoise
- #4660: Avoid extra copy for sparse compressed [:, seq] and [seq, :]...
- #4661: Clean, rebase of #4478, adding ?gelsy and ?gelsd wrappers
- #4662: MAINT: Correct odeint messages
- #4664: Update _monotone.py
- #4672: fix behavior of scipy.linalg.block_diag for empty input
- #4675: Fix lsim
- #4676: Added missing colon to :math: directive in docstring.
- #4679: ENH: sparse randn
- #4682: ENH: scipy.signal - Addition of CSD, coherence; Enhancement of...
- #4684: BUG: various errors in weight calculations in orthogonal.py
- #4685: BUG: Fixes #4594: optimize.linprog IndexError when a callback...
- #4686: MAINT: cluster: Clean up duplicated exception raising code.
- #4688: Improve is_distance_dm exception message
- #4692: MAINT: stats: Simplify the calculation in tukeylambda._ppf
- #4693: ENH: added functionality to handle scalars in *stats._chk_asarray*
- #4694: Vectorization of Anderson-Darling computations.
- #4696: Fix singleton expansion in lfilter.
- #4698: MAINT: quiet warnings from cephes.

- #4701: add Bpoly.antiderivatives / integrals
- #4703: Add citation of published paper
- #4706: MAINT: special: avoid out-of-bounds access in specfun
- #4707: MAINT: fix issues with np.matrix as input to functions related...
- #4709: ENH: `scipy.stats` now returns namedtuples.
- #4710: scipy.io.idl: make reader more robust to missing variables in...
- #4711: Fix crash for unknown chunks at the end of file
- #4712: Reduce onenormest memory usage
- #4713: MAINT: interpolate: no need to pass dtype around if it can be...
- #4714: BENCH: Add benchmarks for stats module
- #4715: MAINT: polish signal.place_poles and signal/test_ltisys.py
- #4716: DEP: deprecate mstats.signaltonoise ...
- #4717: MAINT: basinhopping: fix error in tests, silence /0 warning,...
- #4718: ENH: stats: can specify f-shapes to fix in fitting by name
- #4721: Document that imresize converts the input to a PIL image
- #4722: MAINT: PyArray_BASE is not an lvalue unless the deprecated API...
- #4725: Fix gengamma _nump failure
- #4728: DOC: add poch to the list of scipy special function descriptions
- #4735: MAINT: stats: avoid (a spurious) division-by-zero in skew
- #4738: TST: silence runtime warnings for some corner cases in *stats*...
- #4739: BLD: try to build numpy instead of using the one on TravisCI
- #4740: DOC: Update some docstrings with 'versionadded'.
- #4742: BLD: make sure that relaxed strides checking is in effect on...
- #4750: DOC: special: TeX typesetting of rel_entr, kl_div and pseudo_huber
- #4751: BENCH: add sparse null slice benchmark
- #4753: BUG: Fixed compilation with recent Cython versions.
- #4756: BUG: Fixes #4733: optimize.brute finish option is not compatible...
- #4758: DOC: optimize.leastsq default maxfev clarification
- #4759: improved stats mle fit
- #4760: MAINT: count bfgs updates more carefully
- #4762: BUGS: Fixes #4746 and #4594: linprog returns solution violating...
- #4763: fix small linprog bugs
- #4766: BENCH: add signal.lsim benchmark
- #4768: fix python syntax errors in docstring examples
- #4769: Fixes #4726: test_cobyla.test_vector_constraints
- #4770: Mark FITPACK functions as thread safe.

- #4771: edited scipy/stats/stats.py to fix doctest for fisher_exact
- #4773: DOC: update 0.16.0 release notes.
- #4775: DOC: linalg: add funm_psd as a docstring example
- #4778: Use a dictionary for function name synonyms
- #4780: Include apparently-forgotten functions in docs
- #4783: Added many missing special functions to docs
- #4784: add an axis attribute to PPoly and friends
- #4785: Brief note about origin of Lena image
- #4786: DOC: reformat the Methods section of the KDE docstring
- #4787: Add rice cdf and ppf.
- #4792: CI: add a kludge for detecting test failures which try to disguise...
- #4795: Make refguide_check smarter about false positives
- #4797: BUG/TST: numpoints not updated for incremental Voronoi
- #4799: BUG: spatial: Fix a couple edge cases for the Mahalanobis metric...
- #4801: BUG: Fix TypeError in scipy.optimize._trust-region.py when disp=True.
- #4803: Issues with relaxed strides in QR updating routines
- #4806: MAINT: use an informed initial guess for cauchy fit
- #4810: PEP8ify codata.py
- #4812: BUG: Relaxed strides cleanup in decomp_update.pyx.in
- #4820: BLD: update Bento build for sgemv fix and install cython blas/lapack...
- #4823: ENH: scipy.signal - Addition of spectrogram function
- #4827: DOC: add csd and coherence to __init__.py
- #4833: BLD: fix issue in linalg `*lange` wrappers for g77 builds.
- #4841: TST: fix test failures in scipy.special with mingw32 due to test...
- #4842: DOC: update site.cfg.example. Mostly taken over from Numpy
- #4845: BUG: signal: Make spectrogram's return values order match the...
- #4849: DOC:Fix error in ode docstring example
- #4856: BUG: fix typo causing memleak

1.9 SciPy 0.15.1 Release Notes

SciPy 0.15.1 is a bug-fix release with no new features compared to 0.15.0.

1.9.1 Issues fixed

- #4413: BUG: Tests too strict, f2py doesn't have to overwrite this array
- #4417: BLD: avoid using NPY_API_VERSION to check not using deprecated...

- #4418: Restore and deprecate scipy.linalg.calc_work

1.10 SciPy 0.15.0 Release Notes

Contents

- *SciPy 0.15.0 Release Notes*
  - *New features*
    - *Linear Programming Interface*
    - *Differential evolution, a global optimizer*
    - `scipy.signal` *improvements*
    - `scipy.integrate` *improvements*
    - `scipy.linalg` *improvements*
    - `scipy.sparse` *improvements*
    - `scipy.special` *improvements*
    - `scipy.sparse.csgraph` *improvements*
    - `scipy.stats` *improvements*
  - *Deprecated features*
  - *Backwards incompatible changes*
    - *scipy.ndimage*
    - *scipy.integrate*
  - *Authors*
    - *Issues closed*
    - *Pull requests*

SciPy 0.15.0 is the culmination of 6 months of hard work. It contains several new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.16.x branch, and on adding new features on the master branch.

This release requires Python 2.6, 2.7 or 3.2-3.4 and NumPy 1.5.1 or greater.

1.10.1 New features

Linear Programming Interface

The new function `scipy.optimize.linprog` provides a generic linear programming similar to the way `scipy.optimize.minimize` provides a generic interface to nonlinear programming optimizers. Currently the only method supported is *simplex* which provides a two-phase, dense-matrix-based simplex algorithm. Callbacks functions are supported, allowing the user to monitor the progress of the algorithm.

Differential evolution, a global optimizer

A new *scipy.optimize.differential_evolution* function has been added to the `optimize` module. Differential Evolution is an algorithm used for finding the global minimum of multivariate functions. It is stochastic in nature (does not use gradient methods), and can search large areas of candidate space, but often requires larger numbers of function evaluations than conventional gradient based techniques.

`scipy.signal` improvements

The function *scipy.signal.max_len_seq* was added, which computes a Maximum Length Sequence (MLS) signal.

`scipy.integrate` improvements

It is now possible to use *scipy.integrate* routines to integrate multivariate ctypes functions, thus avoiding callbacks to Python and providing better performance.

`scipy.linalg` improvements

The function *scipy.linalg.orthogonal_procrustes* for solving the procrustes linear algebra problem was added.

BLAS level 2 functions `her`, `syr`, `her2` and `syr2` are now wrapped in `scipy.linalg`.

`scipy.sparse` improvements

scipy.sparse.linalg.svds can now take a `LinearOperator` as its main input.

`scipy.special` improvements

Values of ellipsoidal harmonic (i.e. Lame) functions and associated normalization constants can be now computed using `ellip_harm`, `ellip_harm_2`, and `ellip_normal`.

New convenience functions `entr`, `rel_entr` `kl_div`, `huber`, and `pseudo_huber` were added.

`scipy.sparse.csgraph` improvements

Routines `reverse_cuthill_mckee` and `maximum_bipartite_matching` for computing reorderings of sparse graphs were added.

`scipy.stats` improvements

Added a Dirichlet multivariate distribution, *scipy.stats.dirichlet*.

The new function *scipy.stats.median_test* computes Mood's median test.

The new function *scipy.stats.combine_pvalues* implements Fisher's and Stouffer's methods for combining p-values.

scipy.stats.describe returns a namedtuple rather than a tuple, allowing users to access results by index or by name.

1.10.2 Deprecated features

The *scipy.weave* module is deprecated. It was the only module never ported to Python 3.x, and is not recommended to be used for new code - use Cython instead. In order to support existing code, `scipy.weave` has been packaged separately: https://github.com/scipy/weave. It is a pure Python package, and can easily be installed with `pip install weave`.

`scipy.special.bessel_diff_formula` is deprecated. It is a private function, and therefore will be removed from the public API in a following release.

`scipy.stats.nanmean`, `nanmedian` and `nanstd` functions are deprecated in favor of their numpy equivalents.

1.10.3 Backwards incompatible changes

The functions *scipy.ndimage.minimum_positions*, *scipy.ndimage.maximum_positions*‘ and `scipy.ndimage.extrema` return positions as ints instead of floats.

The format of banded Jacobians in `scipy.integrate.ode` solvers is changed. Note that the previous documentation of this feature was erroneous.

1.10.4 Authors

- Abject +
- Ankit Agrawal +
- Sylvain Bellemare +
- Matthew Brett
- Christian Brodbeck
- Christian Brueffer
- Lars Buitinck
- Evgeni Burovski
- Pierre de Buyl +
- Greg Caporaso +
- CJ Carey
- Jacob Carey +
- Thomas A Caswell
- Helder Cesar +
- Björn Dahlgren +
- Kevin Davies +
- Yotam Doron +
- Marcos Duarte +
- endolith
- Jesse Engel +
- Rob Falck +

- Corey Farwell +
- Jaime Fernandez del Rio +
- Clark Fitzgerald +
- Tom Flannaghan +
- Chad Fulton +
- Jochen Garcke +
- François Garillot +
- André Gaul
- Christoph Gohlke
- Ralf Gommers
- Alex Griffing
- Blake Griffith
- Olivier Grisel
- Charles Harris
- Trent Hauck +
- Ian Henriksen +
- Jinhyok Heo +
- Matt Hickford +
- Andreas Hilboll
- Danilo Horta +
- David Menéndez Hurtado +
- Gert-Ludwig Ingold
- Thouis (Ray) Jones
- Chris Kerr +
- Carl Kleffner +
- Andreas Kloeckner
- Thomas Kluyver +
- Adrian Kretz +
- Johannes Kulick +
- Eric Larson
- Brianna Laugher +
- Denis Laxalde
- Antony Lee +
- Gregory R. Lee +
- Brandon Liu
- Alex Loew +

- Loïc Estève +
- Jaakko Luttinen +
- Benny Malengier
- Tobias Megies +
- Sturla Molden
- Eric Moore
- Brett R. Murphy +
- Paul Nation +
- Andrew Nelson
- Brian Newsom +
- Joel Nothman
- Sergio Oller +
- Janani Padmanabhan +
- Tiago M.D. Pereira +
- Nicolas Del Piano +
- Manuel Reinhardt +
- Thomas Robitaille
- Mike Romberg +
- Alex Rothberg +
- Sebastian Pölsterl +
- Maximilian Singh +
- Brigitta Sipocz +
- Alex Stewart +
- Julian Taylor
- Collin Tokheim +
- James Tomlinson +
- Benjamin Trendelkamp-Schroer +
- Richard Tsai
- Alexey Umnov +
- Jacob Vanderplas
- Joris Vankerschaver
- Bastian Venthur +
- Pauli Virtanen
- Stefan van der Walt
- Yuxiang Wang +
- James T. Webber

- Warren Weckesser
- Axl West +
- Nathan Woods
- Benda Xu +
- Víctor Zabalza +
- Tiziano Zito +

A total of 99 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed

- #1431: ellipk(x) extending its domain for x<0 (Trac #904)
- #1727: consistency of std interface (Trac #1200)
- #1851: Shape parameter negated in genextreme (relative to R, MATLAB,...
- #1889: interp2d is weird (Trac #1364)
- #2188: splev gives wrong values or crashes outside of support when der...
- #2343: scipy.insterpolate's splrep function fails with certain combinations...
- #2669: .signal.ltisys.ss2tf should only apply to MISO systems in current...
- #2911: interpolate.splder() failure on Fedora
- #3171: future of weave in scipy
- #3176: Suggestion to improve error message in scipy.integrate.odeint
- #3198: pdf() and logpdf() methods for scipy.stats.gaussian_kde
- #3318: Travis CI is breaking on test("full")
- #3329: scipy.stats.scoreatpercentile backward-incompatible change not...
- #3362: Reference cycle in scipy.sparse.linalg.eigs with shift-invert...
- #3364: BUG: linalg.hessenberg broken (wrong results)
- #3376: stats f_oneway needs floats
- #3379: Installation of scipy 0.13.3 via zc.buildout fails
- #3403: hierarchy.linkage raises an ugly exception for a compressed 2x2...
- #3422: optimize.curve_fit() handles NaN by returning all parameters...
- #3457: linalg.fractional_matrix_power has no docstring
- #3469: DOC: *ndimage.find_object* ignores zero-values
- #3491: optimize.leastsq() documentation should mention it does not work...
- #3499: cluster.vq.whiten return nan for all zeros column in observations
- #3503: minimize attempts to do vector addition when numpy arrays are...
- #3508: exponweib.logpdf fails for valid parameters
- #3509: libatlas3-base-dev does not exist

- #3550: BUG: anomalous values computed by special.ellipkinc
- #3555: `scipy.ndimage` positions are float instead of int
- #3557: UnivariateSpline.__call__ should pass all relevant args through...
- #3569: No license statement for test data imported from boost?
- #3576: mstats test failure (too sensitive?)
- #3579: Errors on scipy 0.14.x branch using MKL, Ubuntu 14.04 x86_64
- #3580: Operator overloading with sparse matrices
- #3587: Wrong alphabetical order in continuous statistical distribution...
- #3596: scipy.signal.fftconvolve no longer threadsafe
- #3623: BUG: signal.convolve takes longer than it needs to
- #3655: Integer returned from integer data in scipy.signal.periodogram...
- #3662: Travis failure on Numpy 1.5.1 (not reproducible?)
- #3668: dendogram(orientation='foo')
- #3669: KroghInterpolator doesn't pass through points
- #3672: Inserting a knot in a spline
- #3682: misleading documentation of scipy.optimize.curve_fit
- #3699: BUG?: minor problem with scipy.signal.lfilter w/initial conditions
- #3700: Inconsistent exceptions raised by scipy.io.loadmat
- #3703: TypeError for RegularGridInterpolator with big-endian data
- #3714: Misleading error message in eigsh: k must be between 1 and rank(A)-1
- #3720: coo_matrix.setdiag() fails
- #3740: Scipy.Spatial.KdTree (Query) Return Type?
- #3761: Invalid result from scipy.special.btdtri
- #3784: DOC - Special Functions - Drum example fix for higher modes
- #3785: minimize() should have friendlier args=
- #3787: BUG: signal: Division by zero in lombscargle
- #3800: BUG: scipy.sparse.csgraph.shortest_path overwrites input matrix
- #3817: Warning in calculating moments from Binomial distribution for...
- #3821: review scipy usage of *np.ma.is_masked*
- #3829: Linear algebra function documentation doesn't mention default...
- #3830: A bug in Docstring of scipy.linalg.eig
- #3844: Issue with shape parameter returned by genextreme
- #3858: "ImportError: No module named Cython.Compiler.Main" on install
- #3876: savgol_filter not in release notes and has no versionadded
- #3884: scipy.stats.kendalltau empty array error
- #3895: ValueError: illegal value in 12-th argument of internal gesdd...

- #3898: skimage test broken by minmax filter change
- #3901: scipy sparse errors with numpy master
- #3905: DOC: optimize: linprog docstring has two "Returns" sections
- #3915: DOC: sphinx warnings because of **kwds* in the stats distributions...
- #3935: Split stats.distributions files in tutorial
- #3969: gh-3607 breaks backward compatibility in ode solver banded jacobians
- #4025: DOC: signal: The return value of find_peaks_cwt is not documented.
- #4029: scipy.stats.nbinom.logpmf(0,1,1) returns nan. Correct value is...
- #4032: ERROR: test_imresize (test_pilutil.TestPILUtil)
- #4038: errors do not propagate through scipy.integrate.odeint properly
- #4171: orthogonal_procrustes always returns scale.
- #4176: Solving the Discrete Lyapunov Equation does not work with matrix...

Pull requests

- #3109: ENH Added Fisher's method and Stouffer's Z-score method
- #3225: Add the limiting distributions to generalized Pareto distribution...
- #3262: Implement back end of faster multivariate integration
- #3266: ENH: signal: add type=False as parameter for periodogram and...
- #3273: Add PEP8 check to Travis-CI
- #3342: ENH: linprog function for linear programming
- #3348: BUG: add proper error handling when using interp2d on regular...
- #3351: ENH: Add MLS method
- #3382: ENH: scipy.special information theory functions
- #3396: ENH: improve stats.nanmedian more by assuming nans are rare
- #3398: Added two wrappers to the gaussian_kde class.
- #3405: BUG: cluster.linkage array conversion to double dtype
- #3407: MAINT: use assert_warns instead of a more complicated mechanism
- #3409: ENH: change to use array view in signal/_peak_finding.py
- #3416: Issue 3376 : stats f_oneway needs floats
- #3419: BUG: tools: Fix list of FMA instructions in detect_cpu_extensions_wine.py
- #3420: DOC: stats: Add 'entropy' to the stats package-level documentation.
- #3429: BUG: close intermediate file descriptor right after it is used...
- #3430: MAINT: Fix some cython variable declarations to avoid warnings...
- #3433: Correcting the normalization of chebwin window function
- #3435: Add more precise link to R's quantile documentation
- #3446: ENH: scipy.optimize - adding differential_evolution

- #3450: MAINT: remove unused function scipy.stats.mstats_basic._kolmog1
- #3458: Reworked version of PR-3084 (mstats-stats comparison)
- #3462: MAINT : Returning a warning for low attenuation values of chebwin...
- #3463: DOC: linalg: Add examples to functions in matfuncs.py
- #3477: ENH: sparse: release GIL in sparsetools routines
- #3480: DOC: Add more details to deconvolve docstring
- #3484: BLD: fix Qhull build issue with MinGW-w64. Closes gh-3237.
- #3498: MAINT: io: remove old warnings from idl.py
- #3504: BUG: cluster.vq.whiten returns nan or inf when std==0
- #3510: MAINT: stats: Reimplement the pdf and logpdf methods of exponweib.
- #3512: Fix PEP8 errors showing up on TravisCI after pep8 1.5 release
- #3514: DOC: libatlas3-base-dev seems to have never been a thing
- #3516: DOC improve scipy.sparse docstrings
- #3517: ENH: speed-up ndimage.filters.min(max)imum_filter1d
- #3518: Issues in scipy.misc.logsumexp
- #3526: DOC: graphical example for cwt, and use a more interesting signal
- #3527: ENH: Implement min(max)imum_filter1d using the MINLIST algorithm
- #3537: STY: reduce number of C compiler warnings
- #3540: DOC: linalg: add docstring to fractional_matrix_power
- #3542: kde.py Doc Typo
- #3545: BUG: stats: stats.levy.cdf with small arguments loses precision.
- #3547: BUG: special: erfcinv with small arguments loses precision.
- #3553: DOC: Convolve examples
- #3561: FIX: in ndimage.measurements return positions as int instead...
- #3564: Fix test failures with numpy master. Closes gh-3554
- #3565: ENH: make interp2d accept unsorted arrays for interpolation.
- #3566: BLD: add numpy requirement to metadata if it can't be imported.
- #3567: DOC: move matfuncs docstrings to user-visible functions
- #3574: Fixes multiple bugs in mstats.theilslopes
- #3577: TST: decrease sensitivity of an mstats test
- #3585: Cleanup of code in scipy.constants
- #3589: BUG: sparse: allow operator overloading
- #3594: BUG: lobpcg returned wrong values for small matrices (n < 10)
- #3598: MAINT: fix coverage and coveralls
- #3599: MAINT: symeig – now that's a name I've not heard in a long time
- #3602: MAINT: clean up the new optimize.linprog and add a few more tests

- #3607: BUG: integrate: Fix some bugs and documentation errors in the...
- #3609: MAINT integrate/odepack: kill dead Fortran code
- #3616: FIX: Invalid values
- #3617: Sort netcdf variables in a Python-3 compatible way
- #3622: DOC: Added 0.15.0 release notes entry for linprog function.
- #3625: Fix documentation for cKDTree.sparse_distance_matrix
- #3626: MAINT: linalg.orth memory efficiency
- #3627: MAINT: stats: A bit of clean up
- #3628: MAINT: signal: remove a useless function from wavelets.py
- #3632: ENH: stats: Add Mood's median test.
- #3636: MAINT: cluster: some clean up
- #3638: DOC: docstring of optimize.basinhopping confuses singular and...
- #3639: BUG: change ddof default to 1 in mstats.sem, consistent with...
- #3640: Weave: deprecate the module and disable slow tests on TravisCI
- #3641: ENH: Added support for date attributes to io.arff.arffread
- #3644: MAINT: stats: remove superfluous alias in mstats_basic.py
- #3646: ENH: adding *sum_duplicates* method to COO sparse matrix
- #3647: Fix for #3596: Make fftconvolve threadsafe
- #3650: BUG: sparse: smarter random index selection
- #3652: fix wrong option name in power_divergence dosctring example
- #3654: Changing EPD to Canopy
- #3657: BUG: signal.welch: ensure floating point dtype regardless of...
- #3660: TST: mark a test as known fail
- #3661: BLD: ignore pep8 E302 (expected 2 blank lines, found 1)
- #3663: BUG: fix leaking errstate, and ignore invalid= errors in a test
- #3664: BUG: correlate was extremely slow when in2.size > in1.size
- #3667: ENH: Adds default params to pdfs of multivariate_norm
- #3670: ENH: Small speedup of FFT size check
- #3671: DOC: adding differential_evolution function to 0.15 release notes
- #3673: BUG: interpolate/fitpack: arguments to fortran routines may not...
- #3674: Add support for appending to existing netcdf files
- #3681: Speed up test('full'), solve Travis CI timeout issues
- #3683: ENH: cluster: rewrite and optimize *vq* in Cython
- #3684: Update special docs
- #3688: Spacing in special docstrings
- #3692: ENH: scipy.special: Improving sph_harm function

- #3693: Update refguide entries for signal and fftpack
- #3695: Update continuous.rst
- #3696: ENH: check for valid 'orientation' kwarg in dendrogram()
- #3701: make 'a' and 'b' coefficients atleast_1d array in filtfilt
- #3702: BUG: cluster: _vq unable to handle large features
- #3704: BUG: special: ellip(k,e)inc nan and double expected value
- #3707: BUG: handle fill_value dtype checks correctly in RegularGridInterpolator
- #3708: Reraise exception on failure to read mat file.
- #3709: BUG: cast 'x' to correct dtype in KroghInterpolator._evaluate
- #3712: ENH: cluster: reimplement the update-step of K-means in Cython
- #3713: FIX: Check type of lfiltic
- #3718: Changed INSTALL file extension to rst
- #3719: address svds returning nans for zero input matrix
- #3722: MAINT: spatial: static, unused code, sqrt(sqeuclidean)
- #3725: ENH: use numpys nanmedian if available
- #3727: TST: add a new fixed_point test and change some test function...
- #3731: BUG: fix romb in scipy.integrate.quadrature
- #3734: DOC: simplify examples with semilogx
- #3735: DOC: Add minimal docstrings to lti.impulse/step
- #3736: BUG: cast pchip arguments to floats
- #3744: stub out inherited methods of Akima1DInterpolator
- #3746: DOC: Fix formatting for Raises section
- #3748: ENH: Added discrete Lyapunov transformation solve
- #3750: Enable automated testing with Python 3.4
- #3751: Reverse Cuthill-McKee and Maximum Bipartite Matching reorderings...
- #3759: MAINT: avoid indexing with a float array
- #3762: TST: filter out RuntimeWarning in vq tests
- #3766: TST: cluster: some cleanups in test_hierarchy.py
- #3767: ENH/BUG: support negative m in elliptic integrals
- #3769: ENH: avoid repeated matrix inverse
- #3770: BUG: signal: In lfilter_zi, b was not rescaled correctly when...
- #3772: STY avoid unnecessary transposes in csr_matrix.getcol/row
- #3773: ENH: Add ext parameter to UnivariateSpline call
- #3774: BUG: in integrate/quadpack.h, put all declarations before statements.
- #3779: Incbet fix
- #3788: BUG: Fix lombscargle ZeroDivisionError

- #3791: Some maintenance for doc builds
- #3795: scipy.special.legendre docstring
- #3796: TYPO: sheroidal -> spheroidal
- #3801: BUG: shortest_path overwrite
- #3803: TST: lombscargle regression test related to atan vs atan2
- #3809: ENH: orthogonal procrustes solver
- #3811: ENH: scipy.special, Implemented Ellipsoidal harmonic function:...
- #3819: BUG: make a fully connected csgraph from an ndarray with no zeros
- #3820: MAINT: avoid spurious warnings in binom(n, p=0).mean() etc
- #3825: Don't claim scipy.cluster does distance matrix calculations.
- #3827: get and set diagonal of coo_matrix, and related csgraph laplacian...
- #3832: DOC: Minor additions to integrate/nquad docstring.
- #3845: Bug fix for #3842: Bug in scipy.optimize.line_search
- #3848: BUG: edge case where the covariance matrix is exactly zero
- #3850: DOC: typo
- #3851: DOC: document default argument values for some arpack functions
- #3860: DOC: sparse: add the function 'find' to the module-level docstring
- #3861: BUG: Removed unnecessary storage of args as instance variables...
- #3862: BUG: signal: fix handling of multi-output systems in ss2tf.
- #3865: Feature request: ability to read heterogeneous types in FortranFile
- #3866: MAINT: update pip wheelhouse for installs
- #3871: MAINT: linalg: get rid of calc_lwork.f
- #3872: MAINT: use scipy.linalg instead of np.dual
- #3873: BLD: show a more informative message if Cython wasn't installed.
- #3874: TST: cluster: cleanup the hierarchy test data
- #3877: DOC: Savitzky-Golay filter version added
- #3878: DOC: move versionadded to notes
- #3879: small tweaks to the docs
- #3881: FIX incorrect sorting during fancy assignment
- #3885: kendalltau function now returns a nan tuple if empty arrays used...
- #3886: BUG: fixing linprog's kwarg order to match docs
- #3888: BUG: optimize: In _linprog_simplex, handle the case where the...
- #3891: BUG: stats: Fix ValueError message in chi2_contingency.
- #3892: DOC: sparse.linalg: Fix lobpcg docstring.
- #3894: DOC: stats: Assorted docstring edits.
- #3896: Fix 2 mistakes in MatrixMarket format parsing

- #3897: BUG: associated Legendre function of second kind for 1<x<1.0001
- #3899: BUG: fix undefined behavior in alngam
- #3906: MAINT/DOC: Whitespace tweaks in several docstrings.
- #3907: TST: relax bounds of interpolate test to accomodate rounding...
- #3909: MAINT: Create a common version of *count_nonzero* for compatibility...
- #3910: Fix a couple of test errors in master
- #3911: Use MathJax for the html docs
- #3914: Rework the _roots functions and document them.
- #3916: Remove all linpack_lite code and replace with LAPACK routines
- #3917: splines, constant extrapolation
- #3918: DOC: tweak the rv_discrete docstring example
- #3919: Quadrature speed-up: scipy.special.orthogonal.p_roots with cache
- #3920: DOC: Clarify docstring for *sigma* parameter for *curve_fit*
- #3922: Fixed Docstring issues in linprog (Fixes #3905).
- #3924: Coerce args into tuple if necessary.
- #3926: DOC: Surround stats class methods in docstrings with backticks.
- #3927: Changed doc for romb's dx parameter to int.
- #3928: check FITPACK conditions in LSQUnivariateSpline
- #3929: Added a warning about leastsq using with NaNs.
- #3930: ENH: optimize: curve_fit now warns if pcov is undetermined
- #3932: Clarified the k > n case.
- #3933: DOC: remove *import scipy as sp* abbreviation here and there
- #3936: Add license and copyright holders to test data imported from...
- #3938: DOC: Corrected documentation for return types.
- #3939: DOC: fitpack: add a note about Sch-W conditions to splrep docstring
- #3940: TST: integrate: Remove an invalid test of odeint.
- #3942: FIX: Corrected error message of eigsh.
- #3943: ENH: release GIL for filter and interpolation of ndimage
- #3944: FIX: Raise value error if window data-type is unsupported
- #3946: Fixed signal.get_window with unicode window name
- #3947: MAINT: some docstring fixes and style cleanups in stats.mstats
- #3949: DOC: fix a couple of issues in stats docstrings.
- #3950: TST: sparse: remove known failure that doesn't fail
- #3951: TST: switch from Rackspace wheelhouse to numpy/cython source...
- #3952: DOC: stats: Small formatting correction to the 'chi' distribution...
- #3953: DOC: stats: Several corrections and small additions to docstrings.

- #3955: signal.__init__.py: remove duplicated *get_window* entry
- #3959: TST: sparse: more "known failures" for DOK that don't fail
- #3960: BUG: io.netcdf: do not close mmap if there are references left...
- #3965: DOC: Fix a few more sphinx warnings that occur when building...
- #3966: DOC: add guidelines for using test generators in HACKING
- #3968: BUG: sparse.linalg: make Inv objects in arpack garbage-collectable...
- #3971: Remove all linpack_lite code and replace with LAPACK routines
- #3972: fix typo in error message
- #3973: MAINT: better error message for multivariate normal.
- #3981: turn the cryptically named scipy.special information theory functions...
- #3984: Wrap her, syr, her2, syr2 blas routines
- #3990: improve UnivariateSpline docs
- #3991: ENH: stats: return namedtuple for describe output
- #3993: DOC: stats: percentileofscore references np.percentile
- #3997: BUG: linalg: pascal(35) was incorrect: last element overflowed...
- #3998: MAINT: use isMaskedArray instead of is_masked to check type
- #3999: TST: test against all of boost data files.
- #4000: BUG: stats: Fix edge-case handling in a few distributions.
- #4003: ENH: using python's warnings instead of prints in fitpack.
- #4004: MAINT: optimize: remove a couple unused variables in zeros.c
- #4006: BUG: Fix C90 compiler warnings in *NI_MinOrMaxFilter1D*
- #4007: MAINT/DOC: Fix spelling of 'decomposition' in several files.
- #4008: DOC: stats: Split the descriptions of the distributions in the...
- #4015: TST: logsumexp regression test
- #4016: MAINT: remove some inf-related warnings from logsumexp
- #4020: DOC: stats: fix whitespace in docstrings of several distributions
- #4023: Exactly one space required before assignments
- #4024: In dendrogram(): Correct an argument name and a grammar issue...
- #4041: BUG: misc: Ensure that the 'size' argument of PIL's 'resize'...
- #4049: BUG: Return of _logpmf
- #4051: BUG: expm of integer matrices
- #4052: ENH: integrate: odeint: Handle exceptions in the callback functions.
- #4053: BUG: stats: Refactor argument validation to avoid a unicode issue.
- #4057: Added newline to scipy.sparse.linalg.svds documentation for correct...
- #4058: MAINT: stats: Add note about change to scoreatpercentile in release...
- #4059: ENH: interpolate: Allow splev to accept an n-dimensional array.

- #4064: Documented the return value for scipy.signal.find_peaks_cwt
- #4074: ENH: Support LinearOperator as input to svds
- #4084: BUG: Match exception declarations in scipy/io/matlab/streams.pyx...
- #4091: DOC: special: more clear instructions on how to evaluate polynomials
- #4105: BUG: Workaround for SGEMV segfault in Accelerate
- #4107: DOC: get rid of 'import *' in examples
- #4113: DOC: fix typos in distance.yule
- #4114: MAINT C fixes
- #4117: deprecate nanmean, nanmedian and nanstd in favor of their numpy...
- #4126: scipy.io.idl: support description records and fix bug with null...
- #4131: ENH: release GIL in more ndimage functions
- #4132: MAINT: stats: fix a typo [skip ci]
- #4145: DOC: Fix documentation error for nc chi-squared dist
- #4150: Fix _nd_image.geometric_transform endianness bug
- #4153: MAINT: remove use of deprecated numpy API in lib/lapack/ f2py...
- #4156: MAINT: optimize: remove dead code
- #4159: MAINT: optimize: clean up Zeros code
- #4165: DOC: add missing special functions to __doc__
- #4172: DOC: remove misleading procrustes docstring line
- #4175: DOC: sparse: clarify CSC and CSR constructor usage
- #4177: MAINT: enable np.matrix inputs to solve_discrete_lyapunov
- #4179: TST: fix an intermittently failing test case for special.legendre
- #4181: MAINT: remove unnecessary null checks before free
- #4182: Ellipsoidal harmonics
- #4183: Skip Cython build in Travis-CI
- #4184: Pr 4074
- #4187: Pr/3923
- #4190: BUG: special: fix up ellip_harm build
- #4193: BLD: fix msvc compiler errors
- #4194: BUG: fix buffer dtype mismatch on win-amd64
- #4199: ENH: Changed scipy.stats.describe output from datalen to nobs
- #4201: DOC: add blas2 and nan* deprecations to the release notes
- #4243: TST: bump test tolerances

1.11 SciPy 0.14.1 Release Notes

SciPy 0.14.1 is a bug-fix release with no new features compared to 0.14.0.

1.11.1 Issues closed

- #3630: NetCDF reading results in a segfault
- #3631: SuperLU object not working as expected for complex matrices
- #3733: segfault from map_coordinates
- #3780: Segfault when using CSR/CSC matrix and uint32/uint64
- #3781: BUG: sparse: fix omitted types in sparsetools typemaps
- #3802: 0.14.0 API breakage: _gen generators are missing from scipy.stats.distributions API
- #3805: ndimage test failures with numpy 1.10
- #3812: == sometimes wrong on csr_matrix
- #3853: Many scipy.sparse test errors/failures with numpy 1.9.0b2
- #4084: fix exception declarations for Cython 0.21.1 compatibility
- #4093: BUG: fitpack: avoid a memory error in splev(x, tck, der=k)
- #4104: BUG: Workaround SGEMV segfault in Accelerate (maintenance 0.14.x)
- #4143: BUG: fix ndimage functions for large data
- #4149: Bug in expm for integer arrays
- #4154: Backport gh-4041 for 0.14.1 (Ensure that the 'size' argument of PIL's 'resize' method is a tuple)
- #4163: Backport #4142 (ZeroDivisionError in scipy.sparse.linalg.lsqr)
- #4164: Backport gh-4153 (remove use of deprecated numpy API in lib/lapack/ f2py wrapper)
- #4180: backport pil resize support tuple fix
- #4168: Lots of arpack test failures on windows 32 bits with numpy 1.9.1
- #4203: Matrix multiplication in 0.14.x is more than 10x slower compared...
- #4218: attempt to make ndimage interpolation compatible with numpy relaxed...
- #4225: BUG: off-by-one error in PPoly shape checks
- #4248: BUG: optimize: fix issue with incorrect use of closure for slsqp.

1.12 SciPy 0.14.0 Release Notes

Contents

- *SciPy 0.14.0 Release Notes*
  - *New features*
    - `scipy.interpolate` *improvements*

    - *scipy.linalg improvements*
    - *scipy.optimize improvements*
    - *scipy.stats improvements*
    - *scipy.signal improvements*
    - *scipy.special improvements*
    - *scipy.sparse improvements*
  - *Deprecated features*
    - *anneal*
    - *scipy.stats*
    - *scipy.interpolate*
  - *Backwards incompatible changes*
    - *scipy.special.lpmn*
    - *scipy.sparse.linalg*
    - *scipy.stats*
    - *scipy.interpolate*
  - *Other changes*
  - *Authors*
    - *Issues closed*
    - *Pull requests*

SciPy 0.14.0 is the culmination of 8 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.14.x branch, and on adding new features on the master branch.

This release requires Python 2.6, 2.7 or 3.2-3.4 and NumPy 1.5.1 or greater.

1.12.1 New features

`scipy.interpolate` improvements

A new wrapper function *scipy.interpolate.interpn* for interpolation on regular grids has been added. *interpn* supports linear and nearest-neighbor interpolation in arbitrary dimensions and spline interpolation in two dimensions.

Faster implementations of piecewise polynomials in power and Bernstein polynomial bases have been added as *scipy.interpolate.PPoly* and *scipy.interpolate.BPoly*. New users should use these in favor of *scipy.interpolate.PiecewisePolynomial*.

scipy.interpolate.interp1d now accepts non-monotonic inputs and sorts them. If performance is critical, sorting can be turned off by using the new `assume_sorted` keyword.

Functionality for evaluation of bivariate spline derivatives in `scipy.interpolate` has been added.

The new class *scipy.interpolate.Akima1DInterpolator* implements the piecewise cubic polynomial interpolation scheme devised by H. Akima.

Functionality for fast interpolation on regular, unevenly spaced grids in arbitrary dimensions has been added as *scipy.interpolate.RegularGridInterpolator*.

scipy.linalg improvements

The new function *scipy.linalg.dft* computes the matrix of the discrete Fourier transform.

A condition number estimation function for matrix exponential, *scipy.linalg.expm_cond*, has been added.

scipy.optimize improvements

A set of benchmarks for optimize, which can be run with `optimize.bench()`, has been added.

scipy.optimize.curve_fit now has more controllable error estimation via the `absolute_sigma` keyword.

Support for passing custom minimization methods to `optimize.minimize()` and `optimize.minimize_scalar()` has been added, currently useful especially for combining `optimize.basinhopping()` with custom local optimizer routines.

scipy.stats improvements

A new class *scipy.stats.multivariate_normal* with functionality for multivariate normal random variables has been added.

A lot of work on the `scipy.stats` distribution framework has been done. Moment calculations (skew and kurtosis mainly) are fixed and verified, all examples are now runnable, and many small accuracy and performance improvements for individual distributions were merged.

The new function *scipy.stats.anderson_ksamp* computes the k-sample Anderson-Darling test for the null hypothesis that k samples come from the same parent population.

scipy.signal improvements

`scipy.signal.iirfilter` and related functions to design Butterworth, Chebyshev, elliptical and Bessel IIR filters now all use pole-zero ("zpk") format internally instead of using transformations to numerator/denominator format. The accuracy of the produced filters, especially high-order ones, is improved significantly as a result.

The Savitzky-Golay filter was added with the new functions *scipy.signal.savgol_filter* and *scipy.signal.savgol_coeffs*.

The new function *scipy.signal.vectorstrength* computes the vector strength, a measure of phase synchrony, of a set of events.

scipy.special improvements

The functions *scipy.special.boxcox* and *scipy.special.boxcox1p*, which compute the Box-Cox transformation, have been added.

scipy.sparse improvements

- Significant performance improvement in CSR, CSC, and DOK indexing speed.
- When using Numpy >= 1.9 (to be released in MM 2014), sparse matrices function correctly when given to arguments of np.dot, np.multiply and other ufuncs. With earlier Numpy and Scipy versions, the results of such operations are undefined and usually unexpected.
- Sparse matrices are no longer limited to 2^31 nonzero elements. They automatically switch to using 64-bit index data type for matrices containing more elements. User code written assuming the sparse matrices use int32 as the index data type will continue to work, except for such large matrices. Code dealing with larger matrices needs to accept either int32 or int64 indices.

1.12.2 Deprecated features

anneal

The global minimization function *scipy.optimize.anneal* is deprecated. All users should use the *scipy.optimize.basinhopping* function instead.

scipy.stats

randwcdf and randwppf functions are deprecated. All users should use distribution-specific rvs methods instead.

Probability calculation aliases zprob, fprob and ksprob are deprecated. Use instead the sf methods of the corresponding distributions or the special functions directly.

scipy.interpolate

PiecewisePolynomial class is deprecated.

1.12.3 Backwards incompatible changes

lpmn no longer accepts complex-valued arguments. A new function clpmn with uniform complex analytic behavior has been added, and it should be used instead.

Eigenvectors in the case of generalized eigenvalue problem are normalized to unit vectors in 2-norm, rather than following the LAPACK normalization convention.

The deprecated UMFPACK wrapper in scipy.sparse.linalg has been removed due to license and install issues. If available, scikits.umfpack is still used transparently in the spsolve and factorized functions. Otherwise, SuperLU is used instead in these functions.

The deprecated functions glm, oneway and cmedian have been removed from scipy.stats.

stats.scoreatpercentile now returns an array instead of a list of percentiles.

The API for computing derivatives of a monotone piecewise interpolation has changed: if *p* is a PchipInterpolator object, *p.derivative(der)* returns a callable object representing the derivative of *p*. For in-place derivatives use the second argument of the *__call__* method: *p(0.1, der=2)* evaluates the second derivative of *p* at *x=0.1*.

The method *p.derivatives* has been removed.

1.12.4 Other changes

1.12.5 Authors

- Marc Abramowitz +
- Anders Bech Borchersen +
- Vincent Arel-Bundock +
- Petr Baudis +
- Max Bolingbroke
- François Boulogne
- Matthew Brett
- Lars Buitinck
- Evgeni Burovski
- CJ Carey +
- Thomas A Caswell +
- Pawel Chojnacki +
- Phillip Cloud +
- Stefano Costa +
- David Cournapeau
- David Menendez Hurtado +
- Matthieu Dartiailh +
- Christoph Deil +
- Jörg Dietrich +
- endolith
- Francisco de la Peña +
- Ben FrantzDale +
- Jim Garrison +
- André Gaul
- Christoph Gohlke
- Ralf Gommers
- Robert David Grant
- Alex Griffing
- Blake Griffith
- Yaroslav Halchenko
- Andreas Hilboll
- Kat Huang
- Gert-Ludwig Ingold

- James T. Webber +
- Dorota Jarecka +
- Todd Jennings +
- Thouis (Ray) Jones
- Juan Luis Cano Rodríguez
- ktritz +
- Jacques Kvam +
- Eric Larson +
- Justin Lavoie +
- Denis Laxalde
- Jussi Leinonen +
- lemonlaug +
- Tim Leslie
- Alain Leufroy +
- George Lewis +
- Max Linke +
- Brandon Liu +
- Benny Malengier +
- Matthias Kümmerer +
- Cimarron Mittelsteadt +
- Eric Moore
- Andrew Nelson +
- Niklas Hambüchen +
- Joel Nothman +
- Clemens Novak
- Emanuele Olivetti +
- Stefan Otte +
- peb +
- Josef Perktold
- pjwerneck
- poolio
- Jérôme Roy +
- Carl Sandrock +
- Andrew Sczesnak +
- Shauna +
- Fabrice Silva

- Daniel B. Smith
- Patrick Snape +
- Thomas Spura +
- Jacob Stevenson
- Julian Taylor
- Tomas Tomecek
- Richard Tsai
- Jacob Vanderplas
- Joris Vankerschaver +
- Pauli Virtanen
- Warren Weckesser

A total of 80 people contributed to this release. People with a "+" by their names contributed a patch for the first time. This list of names is automatically generated, and may not be fully complete.

Issues closed

- #1325: add custom axis keyword to dendrogram function in scipy.cluster.hierarchy...
- #1437: Wrong pochhammer symbol for negative integers (Trac #910)
- #1555: scipy.io.netcdf leaks file descriptors (Trac #1028)
- #1569: sparse matrix failed with element-wise multiplication using numpy.multiply()...
- #1833: Sparse matrices are limited to 2^32 non-zero elements (Trac #1307)
- #1834: scipy.linalg.eig does not normalize eigenvector if B is given...
- #1866: stats for invgamma (Trac #1340)
- #1886: stats.zipf floating point warnings (Trac #1361)
- #1887: Stats continuous distributions - floating point warnings (Trac...
- #1897: scoreatpercentile() does not handle empty list inputs (Trac #1372)
- #1918: splint returns incorrect results (Trac #1393)
- #1949: kurtosistest fails in mstats with type error (Trac #1424)
- #2092: scipy.test leaves darwin27compiled_catalog, cpp and so files...
- #2106: stats ENH: shape parameters in distribution docstrings (Trac...
- #2123: Bad behavior of sparse matrices in a binary ufunc (Trac #1598)
- #2152: Fix mmio/fromfile on gzip on Python 3 (Trac #1627)
- #2164: stats.rice.pdf(x, 0) returns nan (Trac #1639)
- #2169: scipy.optimize.fmin_bfgs not handling functions with boundaries...
- #2177: scipy.cluster.hierarchy.ClusterNode.pre_order returns IndexError...
- #2179: coo.todense() segfaults (Trac #1654)
- #2185: Precision of scipy.ndimage.gaussian_filter*() limited (Trac #1660)

- #2186: scipy.stats.mstats.kurtosistest crashes on 1d input (Trac #1661)
- #2238: Negative p-value on hypergeom.cdf (Trac #1719)
- #2283: ascending order in interpolation routines (Trac #1764)
- #2288: mstats.kurtosistest is incorrectly converting to float, and fails...
- #2396: lpmn wrong results for `|z| > 1` (Trac #1877)
- #2398: ss2tf returns num as 2D array instead of 1D (Trac #1879)
- #2406: linkage does not take Unicode strings as method names (Trac #1887)
- #2443: IIR filter design should not transform to tf representation internally
- #2572: class method solve of splu return object corrupted or falsely...
- #2667: stats endless loop ?
- #2671: .stats.hypergeom documentation error in the note about pmf
- #2691: BUG scipy.linalg.lapack: potrf/ptroi interpret their 'lower'...
- #2721: Allow use of ellipsis in scipy.sparse slicing
- #2741: stats: deprecate and remove alias for special functions
- #2742: stats add rvs to rice distribution
- #2765: bugs stats entropy
- #2832: argrelextrema returns tuple of 2 empty arrays when no peaks found...
- #2861: scipy.stats.scoreatpercentile broken for vector *per*
- #2891: COBYLA successful termination when constraints violated
- #2919: test failure with the current master
- #2922: ndimage.percentile_filter ignores origin argument for multidimensional...
- #2938: Sparse/dense matrix inplace operations fail due to __numpy_ufunc__
- #2944: MacPorts builds yield 40Mb worth of build warnings
- #2945: FAIL: test_random_complex (test_basic.TestDet)
- #2947: FAIL: Test some trivial edge cases for savgol_filter()
- #2953: Scipy Delaunay triangulation is not oriented
- #2971: scipy.stats.mstats.winsorize documentation error
- #2980: Problems running what seems a perfectly valid example
- #2996: entropy for rv_discrete is incorrect?!
- #2998: Fix numpy version comparisons
- #3002: python setup.py install fails
- #3014: Bug in stats.fisher_exact
- #3030: relative entropy using scipy.stats.distribution.entropy when...
- #3037: scipy.optimize.curve_fit leads to unexpected behavior when input...
- #3047: mstats.ttest_rel axis=None, requires masked array
- #3059: BUG: Slices of sparse matrices return incorrect dtype

- #3063: range keyword in binned_statistics incorrect
- #3067: cumtrapz not working as expected
- #3069: sinc
- #3086: standard error calculation inconsistent between 'stats' and 'mstats'
- #3094: Add a *perm* function into `scipy.misc` and an enhancement of...
- #3111: scipy.sparse.[hv]stack don't respect anymore the dtype parameter
- #3172: optimize.curve_fit uses different nomenclature from optimize.leastsq
- #3196: scipy.stats.mstats.gmean does not actually take dtype
- #3212: Dot product of csr_matrix causes segmentation fault
- #3227: ZeroDivisionError in broyden1 when initial guess is the right...
- #3238: lbfgsb output not suppressed by disp=0
- #3249: Sparse matrix min/max/etc don't support axis=-1
- #3251: cdist performance issue with 'sqeuclidean' metric
- #3279: logm fails for singular matrix
- #3285: signal.chirp(method='hyp') disallows hyperbolic upsweep
- #3299: MEMORY LEAK: fmin_tnc
- #3330: test failures with the current master
- #3345: scipy and/or numpy change is causing tests to fail in another...
- #3363: splu does not work for non-vector inputs
- #3385: expit does not handle large arguments well
- #3395: specfun.f doesn't compile with MinGW
- #3399: Error message bug in scipy.cluster.hierarchy.linkage
- #3404: interpolate._ppoly doesn't build with MinGW
- #3412: Test failures in signal
- #3466: `` `scipy.sparse.csgraph.shortest_path` `` does not work on `` `scipy.sparse.csr_matrix` `` or `` `lil_matrix` ``

Pull requests

- #442: ENH: sparse: enable 64-bit index arrays & nnz > 2**31
- #2766: DOC: remove doc/seps/technology-preview.rst
- #2772: TST: stats: Added a regression test for stats.wilcoxon. Closes...
- #2778: Clean up stats._support, close statistics review issues
- #2792: BUG io: fix file descriptor closing for netcdf variables
- #2847: Rice distribution: extend to b=0, add an explicit rvs method.
- #2878: [stats] fix formulas for higher moments of dweibull distribution
- #2904: ENH: moments for the zipf distribution

- #2907: ENH: add coverage info with coveralls.io for Travis runs.
- #2932: BUG+TST: setdiag implementation for dia_matrix (Close #2931)...
- #2942: Misc fixes pointed out by Eclipse PyDev static code analysis
- #2946: ENH: allow non-monotonic input in interp1d
- #2986: BUG: runtests: chdir away from root when running tests
- #2987: DOC: linalg: don't recommend np.linalg.norm
- #2992: ENH: Add "limit" parameter to dijkstra calculation
- #2995: ENH: Use int shape
- #3006: DOC: stats: add a log base note to the docstring
- #3007: DEP: stats: Deprecate randwppf and randwcdf
- #3008: Fix mstats.kurtosistest, and test coverage for skewtest/normaltest
- #3009: Minor reST typo
- #3010: Add *scipy.optimize.Result* to API docs
- #3012: Corrects documentation error
- #3052: PEP-8 conformance improvements
- #3064: Binned statistic
- #3068: Fix Issue #3067 fix cumptrapz that was raising an exception when...
- #3073: Arff reader with nominal value of 1 character
- #3074: Some maintenance work
- #3080: Review and clean up all Box-Cox functions
- #3083: Bug: should return 0 if no regions found
- #3085: BUG: Use zpk in IIR filter design to improve accuracy
- #3101: refactor stats tests a bit
- #3112: ENH: implement Akima interpolation in 1D
- #3123: MAINT: an easier way to make ranges from slices
- #3124: File object support for imread and imsave
- #3126: pep8ify stats/distributions.py
- #3134: MAINT: split distributions.py into three files
- #3138: clean up tests for discrete distributions
- #3155: special: handle the edge case lambda=0 in pdtr, pdtrc and pdtrik
- #3156: Rename optimize.Result to OptimizeResult
- #3166: BUG: make curve_fit() work with array_like input. Closes gh-3037.
- #3170: Fix numpy version checks
- #3175: use numpy sinc
- #3177: Update numpy version warning, remove oldnumeric import
- #3178: DEP: remove deprecated umfpack wrapper. Closes gh-3002.

- #3179: DOC: add BPoly to the docs
- #3180: Suppress warnings when running stats.test()
- #3181: altered sem func in mstats to match stats
- #3182: Make weave tests behave
- #3183: ENH: Add k-sample Anderson-Darling test to stats module
- #3186: Fix stats.scoreatpercentile
- #3187: DOC: make curve_fit nomenclature same as leastsq
- #3201: Added axis keyword to dendrogram function
- #3207: Make docstring examples in stats.distributions docstrings runnable
- #3218: BUG: integrate: Fix banded jacobian handling in the "vode" and...
- #3222: BUG: limit input ranges in special.nctdtr
- #3223: Fix test errors with numpy master
- #3224: Fix int32 overflows in sparsetools
- #3228: DOC: tf2ss zpk2ss note controller canonical form
- #3234: Add See Also links and Example graphs to filter design `*ord` functions
- #3235: Updated the buttord function to be consistent with the other...
- #3239: correct doc for pchip interpolation
- #3240: DOC: fix ReST errors in the BPoly docstring
- #3241: RF: check write attr of fileobject without writing
- #3243: a bit of maintanence work in stats
- #3245: BUG/ENH: stats: make frozen distributions hold separate instances
- #3247: ENH function to return nnz per row/column in some sparse matrices
- #3248: ENH much more efficient sparse min/max with axis
- #3252: Fast sqeuclidean
- #3253: FIX support axis=-1 and -2 for sparse reduce methods
- #3254: TST tests for non-canonical input to sparse matrix operations
- #3272: BUG: sparse: fix bugs in dia_matrix.setdiag
- #3278: Also generate a tar.xz when running paver sdist
- #3286: DOC: update 0.14.0 release notes.
- #3289: TST: remove insecure mktemp use in tests
- #3292: MAINT: fix a backwards incompatible change to stats.distributions.__all__
- #3293: ENH: signal: Allow upsweeps of frequency in the 'hyperbolic'...
- #3302: ENH: add dtype arg to stats.mstats.gmean and stats.mstats.hmean
- #3307: DOC: add note about different ba forms in tf2zpk
- #3309: doc enhancements to scipy.stats.mstats.winsorize
- #3310: DOC: clarify matrix vs array in mmio docstrings

- #3314: BUG: fix scipy.io.mmread() of gzipped files under Python3
- #3323: ENH: Efficient interpolation on regular grids in arbitrary dimensions
- #3332: DOC: clean up scipy.special docs
- #3335: ENH: improve nanmedian performance
- #3347: BUG: fix use of np.max in stats.fisher_exact
- #3356: ENH: sparse: speed up LIL indexing + assignment via Cython
- #3357: Fix "imresize does not work with size = int"
- #3358: MAINT: rename AkimaInterpolator to Akima1DInterpolator
- #3366: WHT: sparse: reindent `dsolve/*.c *.h`
- #3367: BUG: sparse/dsolve: fix dense matrix fortran order bugs in superlu...
- #3369: ENH minimize, minimize_scalar: Add support for user-provided...
- #3371: scipy.stats.sigmaclip doesn't appear in the html docs.
- #3373: BUG: sparse/dsolve: detect invalid LAPACK parameters in superlu...
- #3375: ENH: sparse/dsolve: make the L and U factors of splu and spilu...
- #3377: MAINT: make travis build one target against Numpy 1.5
- #3378: MAINT: fftpack: Remove the use of `'import *'` in a couple test...
- #3381: MAINT: replace np.isinf(x) & (x>0) -> np.isposinf(x) to avoid...
- #3383: MAINT: skip float96 tests on platforms without float96
- #3384: MAINT: add pyflakes to Travis-CI
- #3386: BUG: stable evaluation of expit
- #3388: BUG: SuperLU: fix missing declaration of dlamch
- #3389: BUG: sparse: downcast 64-bit indices safely to intp when required
- #3390: BUG: nonlinear solvers are not confused by lucky guess
- #3391: TST: fix sparse test errors due to axis=-1,-2 usage in np.matrix.sum().
- #3392: BUG: sparse/lil: fix up Cython bugs in fused type lookup
- #3393: BUG: sparse/compressed: work around bug in np.unique in earlier...
- #3394: BUG: allow ClusterNode.pre_order() for non-root nodes
- #3400: BUG: cluster.linkage ValueError typo bug
- #3402: BUG: special: In specfun.f, replace the use of CMPLX with DCMPLX,...
- #3408: MAINT: sparse: Numpy 1.5 compatibility fixes
- #3410: MAINT: interpolate: fix blas defs in _ppoly
- #3411: MAINT: Numpy 1.5 fixes in interpolate
- #3413: Fix more test issues with older numpy versions
- #3414: TST: signal: loosen some error tolerances in the filter tests....
- #3415: MAINT: tools: automated close issue + pr listings for release...
- #3440: MAINT: wrap sparsetools manually instead via SWIG

- #3460: TST: open image file in binary mode
- #3467: BUG: fix validation in csgraph.shortest_path

1.13 SciPy 0.13.2 Release Notes

SciPy 0.13.2 is a bug-fix release with no new features compared to 0.13.1.

1.13.1 Issues fixed

- 3096: require Cython 0.19, earlier versions have memory leaks in fused types
- 3079: `ndimage.label` fix swapped 64-bitness test
- 3108: `optimize.fmin_slsqp` constraint violation

1.14 SciPy 0.13.1 Release Notes

SciPy 0.13.1 is a bug-fix release with no new features compared to 0.13.0. The only changes are several fixes in `ndimage`, one of which was a serious regression in `ndimage.label` (Github issue 3025), which gave incorrect results in 0.13.0.

1.14.1 Issues fixed

- 3025: `ndimage.label` returns incorrect results in scipy 0.13.0
- 1992: `ndimage.label` return type changed from int32 to uint32
- 1992: `ndimage.find_objects` doesn't work with int32 input in some cases

1.15 SciPy 0.13.0 Release Notes

Contents

- *SciPy 0.13.0 Release Notes*
  - *New features*
    - `scipy.integrate` *improvements*
      - *N-dimensional numerical integration*
      - `dopri*` *improvements*
    - `scipy.linalg` *improvements*
      - *Interpolative decompositions*
      - *Polar decomposition*
      - *BLAS level 3 functions*

        - *Matrix functions*
    - `scipy.optimize` *improvements*
        - *Trust-region unconstrained minimization algorithms*
    - `scipy.sparse` *improvements*
        - *Boolean comparisons and sparse matrices*
        - *CSR and CSC fancy indexing*
    - `scipy.sparse.linalg` *improvements*
    - `scipy.spatial` *improvements*
    - `scipy.signal` *improvements*
    - `scipy.special` *improvements*
    - `scipy.io` *improvements*
        - *Unformatted Fortran file reader*
        - `scipy.io.wavfile` *enhancements*
    - `scipy.interpolate` *improvements*
        - *B-spline derivatives and antiderivatives*
    - `scipy.stats` *improvements*
- *Deprecated features*
    - `expm2` *and* `expm3`
    - `scipy.stats` *functions*
- *Backwards incompatible changes*
    - *LIL matrix assignment*
    - *Deprecated* `radon` *function removed*
    - *Removed deprecated keywords* `xa` *and* `xb` *from* `stats.distributions`
    - *Changes to MATLAB file readers / writers*
- *Other changes*
- *Authors*

SciPy 0.13.0 is the culmination of 7 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.13.x branch, and on adding new features on the master branch.

This release requires Python 2.6, 2.7 or 3.1-3.3 and NumPy 1.5.1 or greater. Highlights of this release are:

- support for fancy indexing and boolean comparisons with sparse matrices
- interpolative decompositions and matrix functions in the linalg module
- two new trust-region solvers for unconstrained minimization

1.15.1 New features

`scipy.integrate` improvements

N-dimensional numerical integration

A new function *scipy.integrate.nquad*, which provides N-dimensional integration functionality with a more flexible interface than `dblquad` and `tplquad`, has been added.

`dopri*` improvements

The intermediate results from the `dopri` family of ODE solvers can now be accessed by a *solout* callback function.

`scipy.linalg` improvements

Interpolative decompositions

Scipy now includes a new module *scipy.linalg.interpolative* containing routines for computing interpolative matrix decompositions (ID). This feature is based on the ID software package by P.G. Martinsson, V. Rokhlin, Y. Shkolnisky, and M. Tygert, previously adapted for Python in the PymatrixId package by K.L. Ho.

Polar decomposition

A new function *scipy.linalg.polar*, to compute the polar decomposition of a matrix, was added.

BLAS level 3 functions

The BLAS functions `symm`, `syrk`, `syr2k`, `hemm`, `herk` and `her2k` are now wrapped in *scipy.linalg*.

Matrix functions

Several matrix function algorithms have been implemented or updated following detailed descriptions in recent papers of Nick Higham and his co-authors. These include the matrix square root (`sqrtm`), the matrix logarithm (`logm`), the matrix exponential (`expm`) and its Frechet derivative (`expm_frechet`), and fractional matrix powers (`fractional_matrix_power`).

`scipy.optimize` improvements

Trust-region unconstrained minimization algorithms

The `minimize` function gained two trust-region solvers for unconstrained minimization: `dogleg` and `trust-ncg`.

`scipy.sparse` improvements

Boolean comparisons and sparse matrices

All sparse matrix types now support boolean data, and boolean operations. Two sparse matrices A and B can be compared in all the expected ways $A < B$, $A >= B$, $A != B$, producing similar results as dense Numpy arrays. Comparisons with dense matrices and scalars are also supported.

CSR and CSC fancy indexing

Compressed sparse row and column sparse matrix types now support fancy indexing with boolean matrices, slices, and lists. So where A is a (CSC or CSR) sparse matrix, you can do things like:

```
>>> A[A > 0.5] = 1  # since Boolean sparse matrices work
>>> A[:2, :3] = 2
>>> A[[1,2], 2] = 3
```

`scipy.sparse.linalg` improvements

The new function `onenormest` provides a lower bound of the 1-norm of a linear operator and has been implemented according to Higham and Tisseur (2000). This function is not only useful for sparse matrices, but can also be used to estimate the norm of products or powers of dense matrices without explicitly building the intermediate matrix.

The multiplicative action of the matrix exponential of a linear operator (`expm_multiply`) has been implemented following the description in Al-Mohy and Higham (2011).

Abstract linear operators (*scipy.sparse.linalg.LinearOperator*) can now be multiplied, added to each other, and exponentiated, producing new linear operators. This enables easier construction of composite linear operations.

`scipy.spatial` improvements

The vertices of a *ConvexHull* can now be accessed via the *vertices* attribute, which gives proper orientation in 2-D.

`scipy.signal` improvements

The cosine window function *scipy.signal.cosine* was added.

`scipy.special` improvements

New functions *scipy.special.xlogy* and *scipy.special.xlog1py* were added. These functions can simplify and speed up code that has to calculate `x * log(y)` and give 0 when `x == 0`.

`scipy.io` improvements

Unformatted Fortran file reader

The new class *scipy.io.FortranFile* facilitates reading unformatted sequential files written by Fortran code.

***scipy.io.wavfile* enhancements**

scipy.io.wavfile.write now accepts a file buffer. Previously it only accepted a filename.

scipy.io.wavfile.read and *scipy.io.wavfile.write* can now handle floating point WAV files.

`scipy.interpolate` improvements

B-spline derivatives and antiderivatives

scipy.interpolate.splder and *scipy.interpolate.splantider* functions for computing B-splines that represent derivatives and antiderivatives of B-splines were added. These functions are also available in the class-based FITPACK interface as `UnivariateSpline.derivative` and `UnivariateSpline.antiderivative`.

`scipy.stats` improvements

Distributions now allow using keyword parameters in addition to positional parameters in all methods.

The function *scipy.stats.power_divergence* has been added for the Cressie-Read power divergence statistic and goodness of fit test. Included in this family of statistics is the "G-test" (http://en.wikipedia.org/wiki/G-test).

scipy.stats.mood now accepts multidimensional input.

An option was added to *scipy.stats.wilcoxon* for continuity correction.

scipy.stats.chisquare now has an *axis* argument.

scipy.stats.mstats.chisquare now has *axis* and *ddof* arguments.

1.15.2 Deprecated features

`expm2` and `expm3`

The matrix exponential functions `scipy.linalg.expm2` and `scipy.linalg.expm3` are deprecated. All users should use the numerically more robust *scipy.linalg.expm* function instead.

`scipy.stats` functions

scipy.stats.oneway is deprecated; *scipy.stats.f_oneway* should be used instead.

scipy.stats.glm is deprecated. *scipy.stats.ttest_ind* is an equivalent function; more full-featured general (and generalized) linear model implementations can be found in statsmodels.

scipy.stats.cmedian is deprecated; `numpy.median` should be used instead.

1.15.3 Backwards incompatible changes

LIL matrix assignment

Assigning values to LIL matrices with two index arrays now works similarly as assigning into ndarrays:

```
>>> x = lil_matrix((3, 3))
>>> x[[0,1,2],[0,1,2]]=[0,1,2]
>>> x.todense()
matrix([[ 0.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  2.]])
```

rather than giving the result:

```
>>> x.todense()
matrix([[ 0.,  1.,  2.],
        [ 0.,  1.,  2.],
        [ 0.,  1.,  2.]])
```

Users relying on the previous behavior will need to revisit their code. The previous behavior is obtained by `x[numpy.ix_([0,1,2],[0,1,2])] = ...`.

Deprecated `radon` function removed

The `misc.radon` function, which was deprecated in scipy 0.11.0, has been removed. Users can find a more full-featured `radon` function in scikit-image.

Removed deprecated keywords `xa` and `xb` from `stats.distributions`

The keywords `xa` and `xb`, which were deprecated since 0.11.0, have been removed from the distributions in `scipy.stats`.

Changes to MATLAB file readers / writers

The major change is that 1D arrays in numpy now become row vectors (shape 1, N) when saved to a MATLAB 5 format file. Previously 1D arrays saved as column vectors (N, 1). This is to harmonize the behavior of writing MATLAB 4 and 5 formats, and adapt to the defaults of numpy and MATLAB - for example `np.atleast_2d` returns 1D arrays as row vectors.

Trying to save arrays of greater than 2 dimensions in MATLAB 4 format now raises an error instead of silently reshaping the array as 2D.

`scipy.io.loadmat('afile')` used to look for *afile* on the Python system path (`sys.path`); now `loadmat` only looks in the current directory for a relative path filename.

1.15.4 Other changes

Security fix: `scipy.weave` previously used temporary directories in an insecure manner under certain circumstances.

Cython is now required to build *unreleased* versions of scipy. The C files generated from Cython sources are not included in the git repo anymore. They are however still shipped in source releases.

The code base received a fairly large PEP8 cleanup. A `tox pep8` command has been added; new code should pass this test command.

Scipy cannot be compiled with gfortran 4.1 anymore (at least on RH5), likely due to that compiler version not supporting entry constructs well.

1.15.5 Authors

This release contains work by the following people (contributed at least one patch to this release, names in alphabetical order):

- Jorge Cañardo Alastuey +
- Tom Aldcroft +
- Max Bolingbroke +
- Joseph Jon Booker +
- François Boulogne
- Matthew Brett
- Christian Brodbeck +
- Per Brodtkorb +

- Christian Brueffer +
- Lars Buitinck
- Evgeni Burovski +
- Tim Cera
- Lawrence Chan +
- David Cournapeau
- Drazen Lucanin +
- Alexander J. Dunlap +
- endolith
- André Gaul +
- Christoph Gohlke
- Ralf Gommers
- Alex Griffing +
- Blake Griffith +
- Charles Harris
- Bob Helmbold +
- Andreas Hilboll
- Kat Huang +
- Oleksandr (Sasha) Huziy +
- Gert-Ludwig Ingold +
- Thouis (Ray) Jones
- Juan Luis Cano Rodríguez +
- Robert Kern
- Andreas Kloeckner +
- Sytse Knypstra +
- Gustav Larsson +
- Denis Laxalde
- Christopher Lee
- Tim Leslie
- Wendy Liu +
- Clemens Novak +
- Takuya Oshima +
- Josef Perktold
- Illia Polosukhin +
- Przemek Porebski +
- Steve Richardson +

- Branden Rolston +
- Skipper Seabold
- Fazlul Shahriar
- Leo Singer +
- Rohit Sivaprasad +
- Daniel B. Smith +
- Julian Taylor
- Louis Thibault +
- Tomas Tomecek +
- John Travers
- Richard Tsai +
- Jacob Vanderplas
- Patrick Varilly
- Pauli Virtanen
- Stefan van der Walt
- Warren Weckesser
- Pedro Werneck +
- Nils Werner +
- Michael Wimmer +
- Nathan Woods +
- Tony S. Yu +

A total of 65 people contributed to this release. People with a "+" by their names contributed a patch for the first time.

1.16 SciPy 0.12.1 Release Notes

SciPy 0.12.1 is a bug-fix release with no new features compared to 0.12.0. The single issue fixed by this release is a security issue in `scipy.weave`, which was previously using temporary directories in an insecure manner under certain circumstances.

1.17 SciPy 0.12.0 Release Notes

Contents

- *SciPy 0.12.0 Release Notes*
  - *New features*
    - *`scipy.spatial` improvements*
      - *cKDTree feature-complete*

        · *Voronoi diagrams and convex hulls*
        · *Delaunay improvements*
      * *Spectral estimators (`scipy.signal`)*
      * *`scipy.optimize` improvements*
        · *Callback functions in L-BFGS-B and TNC*
        · *Basin hopping global optimization (`scipy.optimize.basinhopping`)*
      * *`scipy.special` improvements*
        · *Revised complex error functions*
        · *Faster orthogonal polynomials*
      * *`scipy.sparse.linalg` features*
      * *Listing Matlab(R) file contents in `scipy.io`*
      * *Documented BLAS and LAPACK low-level interfaces (`scipy.linalg`)*
      * *Polynomial interpolation improvements (`scipy.interpolate`)*
    – *Deprecated features*
      * scipy.lib.lapack
      * fblas *and* cblas
    – *Backwards incompatible changes*
      * *Removal of `scipy.io.save_as_module`*
      * axis *argument added to* `scipy.stats.scoreatpercentile`
    – *Authors*

SciPy 0.12.0 is the culmination of 7 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.12.x branch, and on adding new features on the master branch.

Some of the highlights of this release are:

- Completed QHull wrappers in scipy.spatial.
- cKDTree now a drop-in replacement for KDTree.
- A new global optimizer, basinhopping.
- Support for Python 2 and Python 3 from the same code base (no more 2to3).

This release requires Python 2.6, 2.7 or 3.1-3.3 and NumPy 1.5.1 or greater. Support for Python 2.4 and 2.5 has been dropped as of this release.

1.17.1 New features

`scipy.spatial` improvements

cKDTree feature-complete

Cython version of KDTree, cKDTree, is now feature-complete. Most operations (construction, query, query_ball_point, query_pairs, count_neighbors and sparse_distance_matrix) are between 200 and 1000 times faster in cKDTree than in KDTree. With very minor caveats, cKDTree has exactly the same interface as KDTree, and can be used as a drop-in replacement.

Voronoi diagrams and convex hulls

scipy.spatial now contains functionality for computing Voronoi diagrams and convex hulls using the Qhull library. (Delaunay triangulation was available since Scipy 0.9.0.)

Delaunay improvements

It's now possible to pass in custom Qhull options in Delaunay triangulation. Coplanar points are now also recorded, if present. Incremental construction of Delaunay triangulations is now also possible.

Spectral estimators (`scipy.signal`)

The functions `scipy.signal.periodogram` and `scipy.signal.welch` were added, providing DFT-based spectral estimators.

`scipy.optimize` improvements

Callback functions in L-BFGS-B and TNC

A callback mechanism was added to L-BFGS-B and TNC minimization solvers.

Basin hopping global optimization (`scipy.optimize.basinhopping`)

A new global optimization algorithm. Basinhopping is designed to efficiently find the global minimum of a smooth function.

`scipy.special` improvements

Revised complex error functions

The computation of special functions related to the error function now uses a new Faddeeva library from MIT which increases their numerical precision. The scaled and imaginary error functions `erfcx` and `erfi` were also added, and the Dawson integral `dawsn` can now be evaluated for a complex argument.

Faster orthogonal polynomials

Evaluation of orthogonal polynomials (the `eval_*` routines) in now faster in `scipy.special`, and their `out=` argument functions properly.

`scipy.sparse.linalg` features

- In `scipy.sparse.linalg.spsolve`, the `b` argument can now be either a vector or a matrix.
- `scipy.sparse.linalg.inv` was added. This uses `spsolve` to compute a sparse matrix inverse.

- `scipy.sparse.linalg.expm` was added. This computes the exponential of a sparse matrix using a similar algorithm to the existing dense array implementation in `scipy.linalg.expm`.

Listing Matlab(R) file contents in `scipy.io`

A new function `whosmat` is available in `scipy.io` for inspecting contents of MAT files without reading them to memory.

Documented BLAS and LAPACK low-level interfaces (`scipy.linalg`)

The modules *`scipy.linalg.blas`* and *`scipy.linalg.lapack`* can be used to access low-level BLAS and LAPACK functions.

Polynomial interpolation improvements (`scipy.interpolate`)

The barycentric, Krogh, piecewise and pchip polynomial interpolators in `scipy.interpolate` accept now an `axis` argument.

1.17.2 Deprecated features

scipy.lib.lapack

The module *scipy.lib.lapack* is deprecated. You can use *`scipy.linalg.lapack`* instead. The module *scipy.lib.blas* was deprecated earlier in Scipy 0.10.0.

fblas and *cblas*

Accessing the modules *scipy.linalg.fblas*, *cblas*, *flapack*, *clapack* is deprecated. Instead, use the modules *`scipy.linalg.lapack`* and *`scipy.linalg.blas`*.

1.17.3 Backwards incompatible changes

Removal of `scipy.io.save_as_module`

The function `scipy.io.save_as_module` was deprecated in Scipy 0.11.0, and is now removed.

Its private support modules `scipy.io.dumbdbm_patched` and `scipy.io.dumb_shelve` are also removed.

axis argument added to `scipy.stats.scoreatpercentile`

The function *`scipy.stats.scoreatpercentile`* has been given an *axis* argument. The default argument is *axis=None*, which means the calculation is done on the flattened array. Before this change, *scoreatpercentile* would act as if *axis=0* had been given. Code using *scoreatpercentile* with a multidimensional array will need to add *axis=0* to the function call to preserve the old behavior. (This API change was not noticed until long after the release of 0.12.0.)

1.17.4 Authors

- Anton Akhmerov +
- Alexander Eberspächer +
- Anne Archibald
- Jisk Attema +
- K.-Michael Aye +
- bemasc +
- Sebastian Berg +
- François Boulogne +
- Matthew Brett
- Lars Buitinck
- Steven Byrnes +
- Tim Cera +
- Christian +
- Keith Clawson +
- David Cournapeau
- Nathan Crock +
- endolith
- Bradley M. Froehle +
- Matthew R Goodman
- Christoph Gohlke
- Ralf Gommers
- Robert David Grant +
- Yaroslav Halchenko
- Charles Harris
- Jonathan Helmus
- Andreas Hilboll
- Hugo +
- Oleksandr Huziy
- Jeroen Demeyer +
- Johannes Schönberger +
- Steven G. Johnson +
- Chris Jordan-Squire
- Jonathan Taylor +
- Niklas Kroeger +
- Jerome Kieffer +

- kingson +
- Josh Lawrence
- Denis Laxalde
- Alex Leach +
- Tim Leslie
- Richard Lindsley +
- Lorenzo Luengo +
- Stephen McQuay +
- MinRK
- Sturla Molden +
- Eric Moore +
- mszep +
- Matt Newville +
- Vlad Niculae
- Travis Oliphant
- David Parker +
- Fabian Pedregosa
- Josef Perktold
- Zach Ploskey +
- Alex Reinhart +
- Gilles Rochefort +
- Ciro Duran Santillli +
- Jan Schlueter +
- Jonathan Scholz +
- Anthony Scopatz
- Skipper Seabold
- Fabrice Silva +
- Scott Sinclair
- Jacob Stevenson +
- Sturla Molden +
- Julian Taylor +
- thorstenkranz +
- John Travers +
- True Price +
- Nicky van Foreest
- Jacob Vanderplas

- Patrick Varilly
- Daniel Velkov +
- Pauli Virtanen
- Stefan van der Walt
- Warren Weckesser

A total of 75 people contributed to this release. People with a "+" by their names contributed a patch for the first time.

1.18 SciPy 0.11.0 Release Notes

Contents

- *SciPy 0.11.0 Release Notes*
  - *New features*
    - *Sparse Graph Submodule*
    - `scipy.optimize` *improvements*
      - *Unified interfaces to minimizers*
      - *Unified interface to root finding algorithms*
    - `scipy.linalg` *improvements*
      - *New matrix equation solvers*
      - *QZ and QR Decomposition*
      - *Pascal matrices*
    - *Sparse matrix construction and operations*
    - *LSMR iterative solver*
    - *Discrete Sine Transform*
    - `scipy.interpolate` *improvements*
    - *Binned statistics (*`scipy.stats`*)*
  - *Deprecated features*
  - *Backwards incompatible changes*
    - *Removal of* `scipy.maxentropy`
    - *Minor change in behavior of* `splev`
    - *Behavior of* `scipy.integrate.complex_ode`
    - *Minor change in behavior of T-tests*
  - *Other changes*
  - *Authors*

SciPy 0.11.0 is the culmination of 8 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. Highlights of this release are:

- A new module has been added which provides a number of common sparse graph algorithms.
- New unified interfaces to the existing optimization and root finding functions have been added.

All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Our development attention will now shift to bug-fix releases on the 0.11.x branch, and on adding new features on the master branch.

This release requires Python 2.4-2.7 or 3.1-3.2 and NumPy 1.5.1 or greater.

1.18.1 New features

Sparse Graph Submodule

The new submodule `scipy.sparse.csgraph` implements a number of efficient graph algorithms for graphs stored as sparse adjacency matrices. Available routines are:

- `connected_components` - determine connected components of a graph
- `laplacian` - compute the laplacian of a graph
- `shortest_path` - compute the shortest path between points on a positive graph
- `dijkstra` - use Dijkstra's algorithm for shortest path
- `floyd_warshall` - use the Floyd-Warshall algorithm for shortest path
- `breadth_first_order` - compute a breadth-first order of nodes
- `depth_first_order` - compute a depth-first order of nodes
- `breadth_first_tree` - construct the breadth-first tree from a given node
- `depth_first_tree` - construct a depth-first tree from a given node
- `minimum_spanning_tree` - construct the minimum spanning tree of a graph

`scipy.optimize` improvements

The optimize module has received a lot of attention this release. In addition to added tests, documentation improvements, bug fixes and code clean-up, the following improvements were made:

- A unified interface to minimizers of univariate and multivariate functions has been added.
- A unified interface to root finding algorithms for multivariate functions has been added.
- The L-BFGS-B algorithm has been updated to version 3.0.

Unified interfaces to minimizers

Two new functions `scipy.optimize.minimize` and `scipy.optimize.minimize_scalar` were added to provide a common interface to minimizers of multivariate and univariate functions respectively. For multivariate functions, `scipy.optimize.minimize` provides an interface to methods for unconstrained optimization (*fmin*, *fmin_powell*, *fmin_cg*, *fmin_ncg*, *fmin_bfgs* and *anneal*) or constrained optimization (*fmin_l_bfgs_b*, *fmin_tnc*, *fmin_cobyla* and *fmin_slsqp*). For univariate functions, `scipy.optimize.minimize_scalar` provides an interface to methods for unconstrained and bounded optimization (*brent*, *golden*, *fminbound*). This allows for easier comparing and switching between solvers.

Unified interface to root finding algorithms

The new function `scipy.optimize.root` provides a common interface to root finding algorithms for multivariate functions, embedding *fsolve*, *leastsq* and *nonlin* solvers.

scipy.linalg improvements

New matrix equation solvers

Solvers for the Sylvester equation (scipy.linalg.solve_sylvester, discrete and continuous Lyapunov equations (scipy.linalg.solve_lyapunov, scipy.linalg.solve_discrete_lyapunov) and discrete and continuous algebraic Riccati equations (scipy.linalg.solve_continuous_are, scipy.linalg.solve_discrete_are) have been added to scipy.linalg. These solvers are often used in the field of linear control theory.

QZ and QR Decomposition

It is now possible to calculate the QZ, or Generalized Schur, decomposition using scipy.linalg.qz. This function wraps the LAPACK routines sgges, dgges, cgges, and zgges.

The function scipy.linalg.qr_multiply, which allows efficient computation of the matrix product of Q (from a QR decomposition) and a vector, has been added.

Pascal matrices

A function for creating Pascal matrices, scipy.linalg.pascal, was added.

Sparse matrix construction and operations

Two new functions, scipy.sparse.diags and scipy.sparse.block_diag, were added to easily construct diagonal and block-diagonal sparse matrices respectively.

scipy.sparse.csc_matrix and csr_matrix now support the operations sin, tan, arcsin, arctan, sinh, tanh, arcsinh, arctanh, rint, sign, expm1, log1p, deg2rad, rad2deg, floor, ceil and trunc. Previously, these operations had to be performed by operating on the matrices' data attribute.

LSMR iterative solver

LSMR, an iterative method for solving (sparse) linear and linear least-squares systems, was added as scipy.sparse.linalg.lsmr.

Discrete Sine Transform

Bindings for the discrete sine transform functions have been added to scipy.fftpack.

scipy.interpolate improvements

For interpolation in spherical coordinates, the three classes scipy.interpolate.SmoothSphereBivariateSpline, scipy.interpolate.LSQSphereBivariateSpline, and scipy.interpolate.RectSphereBivariateSpline have been added.

Binned statistics (scipy.stats)

The stats module has gained functions to do binned statistics, which are a generalization of histograms, in 1-D, 2-D and multiple dimensions: scipy.stats.binned_statistic, scipy.stats.binned_statistic_2d and scipy.stats.binned_statistic_dd.

1.18.2 Deprecated features

`scipy.sparse.cs_graph_components` has been made a part of the sparse graph submodule, and renamed to `scipy.sparse.csgraph.connected_components`. Calling the former routine will result in a deprecation warning.

`scipy.misc.radon` has been deprecated. A more full-featured radon transform can be found in scikits-image.

`scipy.io.save_as_module` has been deprecated. A better way to save multiple Numpy arrays is the `numpy.savez` function.

The *xa* and *xb* parameters for all distributions in `scipy.stats.distributions` already weren't used; they have now been deprecated.

1.18.3 Backwards incompatible changes

Removal of `scipy.maxentropy`

The `scipy.maxentropy` module, which was deprecated in the 0.10.0 release, has been removed. Logistic regression in scikits.learn is a good and modern alternative for this functionality.

Minor change in behavior of `splev`

The spline evaluation function now behaves similarly to `interp1d` for size-1 arrays. Previous behavior:

```
>>> from scipy.interpolate import splev, splrep, interp1d
>>> x = [1,2,3,4,5]
>>> y = [4,5,6,7,8]
>>> tck = splrep(x, y)
>>> splev([1], tck)
4.
>>> splev(1, tck)
4.
```

Corrected behavior:

```
>>> splev([1], tck)
array([ 4.])
>>> splev(1, tck)
array(4.)
```

This affects also the `UnivariateSpline` classes.

Behavior of `scipy.integrate.complex_ode`

The behavior of the `y` attribute of `complex_ode` is changed. Previously, it expressed the complex-valued solution in the form:

```
z = ode.y[::2] + 1j * ode.y[1::2]
```

Now, it is directly the complex-valued solution:

```
z = ode.y
```

Minor change in behavior of T-tests

The T-tests `scipy.stats.ttest_ind`, `scipy.stats.ttest_rel` and `scipy.stats.ttest_1samp` have been changed so that 0 / 0 now returns NaN instead of 1.

1.18.4 Other changes

The SuperLU sources in `scipy.sparse.linalg` have been updated to version 4.3 from upstream.

The function `scipy.signal.bode`, which calculates magnitude and phase data for a continuous-time system, has been added.

The two-sample T-test `scipy.stats.ttest_ind` gained an option to compare samples with unequal variances, i.e. Welch's T-test.

`scipy.misc.logsumexp` now takes an optional `axis` keyword argument.

1.18.5 Authors

This release contains work by the following people (contributed at least one patch to this release, names in alphabetical order):

- Jeff Armstrong
- Chad Baker
- Brandon Beacher +
- behrisch +
- borishim +
- Matthew Brett
- Lars Buitinck
- Luis Pedro Coelho +
- Johann Cohen-Tanugi
- David Cournapeau
- dougal +
- Ali Ebrahim +
- endolith +
- Bjørn Forsman +
- Robert Gantner +
- Sebastian Gassner +
- Christoph Gohlke
- Ralf Gommers
- Yaroslav Halchenko
- Charles Harris
- Jonathan Helmus +
- Andreas Hilboll +

- Marc Honnorat +
- Jonathan Hunt +
- Maxim Ivanov +
- Thouis (Ray) Jones
- Christopher Kuster +
- Josh Lawrence +
- Denis Laxalde +
- Travis Oliphant
- Joonas Paalasmaa +
- Fabian Pedregosa
- Josef Perktold
- Gavin Price +
- Jim Radford +
- Andrew Schein +
- Skipper Seabold
- Jacob Silterra +
- Scott Sinclair
- Alexis Tabary +
- Martin Teichmann
- Matt Terry +
- Nicky van Foreest +
- Jacob Vanderplas
- Patrick Varilly +
- Pauli Virtanen
- Nils Wagner +
- Darryl Wally +
- Stefan van der Walt
- Liming Wang +
- David Warde-Farley +
- Warren Weckesser
- Sebastian Werk +
- Mike Wimmer +
- Tony S Yu +

A total of 55 people contributed to this release. People with a "+" by their names contributed a patch for the first time.

1.19 SciPy 0.10.1 Release Notes

Contents

- *SciPy 0.10.1 Release Notes*
  - *Main changes*
  - *Other issues fixed*

SciPy 0.10.1 is a bug-fix release with no new features compared to 0.10.0.

1.19.1 Main changes

The most important changes are:

1. The single precision routines of `eigs` and `eigsh` in `scipy.sparse.linalg` have been disabled (they internally use double precision now).
2. A compatibility issue related to changes in NumPy macros has been fixed, in order to make scipy 0.10.1 compile with the upcoming numpy 1.7.0 release.

1.19.2 Other issues fixed

- #835: stats: nan propagation in stats.distributions
- #1202: io: netcdf segfault
- #1531: optimize: make curve_fit work with method as callable.
- #1560: linalg: fixed mistake in eig_banded documentation.
- #1565: ndimage: bug in ndimage.variance
- #1457: ndimage: standard_deviation does not work with sequence of indexes
- #1562: cluster: segfault in linkage function
- #1568: stats: One-sided fisher_exact() returns $p < 1$ for 0 successful attempts
- #1575: stats: zscore and zmap handle the axis keyword incorrectly

1.20 SciPy 0.10.0 Release Notes

Contents

- *SciPy 0.10.0 Release Notes*
  - *New features*
    - *Bento: new optional build system*
    - *Generalized and shift-invert eigenvalue problems in `scipy.sparse.linalg`*
    - *Discrete-Time Linear Systems (`scipy.signal`)*

    * *Enhancements to `scipy.signal`*
    * *Additional decomposition options (`scipy.linalg`)*
    * *Additional special matrices (`scipy.linalg`)*
    * *Enhancements to `scipy.stats`*
    * *Enhancements to `scipy.special`*
    * *Basic support for Harwell-Boeing file format for sparse matrices*
  - *Deprecated features*
    * *`scipy.maxentropy`*
    * *`scipy.lib.blas`*
    * *Numscons build system*
  - *Backwards-incompatible changes*
  - *Other changes*
  - *Authors*

SciPy 0.10.0 is the culmination of 8 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a limited number of deprecations and backwards-incompatible changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.10.x branch, and on adding new features on the development master branch.

Release highlights:

- Support for Bento as optional build system.
- Support for generalized eigenvalue problems, and all shift-invert modes available in ARPACK.

This release requires Python 2.4-2.7 or 3.1- and NumPy 1.5 or greater.

1.20.1 New features

Bento: new optional build system

Scipy can now be built with Bento. Bento has some nice features like parallel builds and partial rebuilds, that are not possible with the default build system (distutils). For usage instructions see BENTO_BUILD.txt in the scipy top-level directory.

Currently Scipy has three build systems, distutils, numscons and bento. Numscons is deprecated and is planned and will likely be removed in the next release.

Generalized and shift-invert eigenvalue problems in `scipy.sparse.linalg`

The sparse eigenvalue problem solver functions `scipy.sparse.eigs/eigh` now support generalized eigenvalue problems, and all shift-invert modes available in ARPACK.

Discrete-Time Linear Systems (`scipy.signal`)

Support for simulating discrete-time linear systems, including `scipy.signal.dlsim`, `scipy.signal.dimpulse`, and `scipy.signal.dstep`, has been added to SciPy. Conversion of linear systems from continuous-

time to discrete-time representations is also present via the `scipy.signal.cont2discrete` function.

Enhancements to `scipy.signal`

A Lomb-Scargle periodogram can now be computed with the new function `scipy.signal.lombscargle`.

The forward-backward filter function `scipy.signal.filtfilt` can now filter the data in a given axis of an n-dimensional numpy array. (Previously it only handled a 1-dimensional array.) Options have been added to allow more control over how the data is extended before filtering.

FIR filter design with `scipy.signal.firwin2` now has options to create filters of type III (zero at zero and Nyquist frequencies) and IV (zero at zero frequency).

Additional decomposition options (`scipy.linalg`)

A sort keyword has been added to the Schur decomposition routine (`scipy.linalg.schur`) to allow the sorting of eigenvalues in the resultant Schur form.

Additional special matrices (`scipy.linalg`)

The functions `hilbert` and `invhilbert` were added to `scipy.linalg`.

Enhancements to `scipy.stats`

- The *one-sided form* of Fisher's exact test is now also implemented in `stats.fisher_exact`.
- The function `stats.chi2_contingency` for computing the chi-square test of independence of factors in a contingency table has been added, along with the related utility functions `stats.contingency.margins` and `stats.contingency.expected_freq`.

Enhancements to `scipy.special`

The functions `logit(p) = log(p/(1-p))` and `expit(x) = 1/(1+exp(-x))` have been implemented as `scipy.special.logit` and `scipy.special.expit` respectively.

Basic support for Harwell-Boeing file format for sparse matrices

Both read and write are support through a simple function-based API, as well as a more complete API to control number format. The functions may be found in scipy.sparse.io.

The following features are supported:

- Read and write sparse matrices in the CSC format
- Only real, symmetric, assembled matrix are supported (RUA format)

1.20.2 Deprecated features

`scipy.maxentropy`

The maxentropy module is unmaintained, rarely used and has not been functioning well for several releases. Therefore it has been deprecated for this release, and will be removed for scipy 0.11. Logistic regression in scikits.learn is a good alternative for this functionality. The `scipy.maxentropy.logsumexp` function has been moved to `scipy.misc`.

`scipy.lib.blas`

There are similar BLAS wrappers in `scipy.linalg` and `scipy.lib`. These have now been consolidated as `scipy.linalg.blas`, and `scipy.lib.blas` is deprecated.

Numscons build system

The numscons build system is being replaced by Bento, and will be removed in one of the next scipy releases.

1.20.3 Backwards-incompatible changes

The deprecated name *invnorm* was removed from `scipy.stats.distributions`, this distribution is available as *invgauss*.

The following deprecated nonlinear solvers from `scipy.optimize` have been removed:

```
- ``broyden_modified`` (bad performance)
- ``broyden1_modified`` (bad performance)
- ``broyden_generalized`` (equivalent to ``anderson``)
- ``anderson2`` (equivalent to ``anderson``)
- ``broyden3`` (obsoleted by new limited-memory broyden methods)
- ``vackar`` (renamed to ``diagbroyden``)
```

1.20.4 Other changes

`scipy.constants` has been updated with the CODATA 2010 constants.

`__all__` dicts have been added to all modules, which has cleaned up the namespaces (particularly useful for interactive work).

An API section has been added to the documentation, giving recommended import guidelines and specifying which submodules are public and which aren't.

1.20.5 Authors

This release contains work by the following people (contributed at least one patch to this release, names in alphabetical order):

- Jeff Armstrong +
- Matthew Brett
- Lars Buitinck +

- David Cournapeau
- FI$H 2000 +
- Michael McNeil Forbes +
- Matty G +
- Christoph Gohlke
- Ralf Gommers
- Yaroslav Halchenko
- Charles Harris
- Thouis (Ray) Jones +
- Chris Jordan-Squire +
- Robert Kern
- Chris Lasher +
- Wes McKinney +
- Travis Oliphant
- Fabian Pedregosa
- Josef Perktold
- Thomas Robitaille +
- Pim Schellart +
- Anthony Scopatz +
- Skipper Seabold +
- Fazlul Shahriar +
- David Simcha +
- Scott Sinclair +
- Andrey Smirnov +
- Collin RM Stocks +
- Martin Teichmann +
- Jake Vanderplas +
- Gaël Varoquaux +
- Pauli Virtanen
- Stefan van der Walt
- Warren Weckesser
- Mark Wiebe +

A total of 35 people contributed to this release. People with a "+" by their names contributed a patch for the first time.

1.21 SciPy 0.9.0 Release Notes

Contents

- *SciPy 0.9.0 Release Notes*
  - *Python 3*
  - *Scipy source code location to be changed*
  - *New features*
    - *Delaunay tesselations (`scipy.spatial`)*
    - *N-dimensional interpolation (`scipy.interpolate`)*
    - *Nonlinear equation solvers (`scipy.optimize`)*
    - *New linear algebra routines (`scipy.linalg`)*
    - *Improved FIR filter design functions (`scipy.signal`)*
    - *Improved statistical tests (`scipy.stats`)*
  - *Deprecated features*
    - *Obsolete nonlinear solvers (in `scipy.optimize`)*
  - *Removed features*
    - *Old correlate/convolve behavior (in `scipy.signal`)*
    - *`scipy.stats`*
    - *`scipy.sparse`*
    - *`scipy.sparse.linalg.arpack.speigs`*
  - *Other changes*
    - *ARPACK interface changes*

SciPy 0.9.0 is the culmination of 6 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.9.x branch, and on adding new features on the development trunk.

This release requires Python 2.4 - 2.7 or 3.1 - and NumPy 1.5 or greater.

Please note that SciPy is still considered to have "Beta" status, as we work toward a SciPy 1.0.0 release. The 1.0.0 release will mark a major milestone in the development of SciPy, after which changing the package structure or API will be much more difficult. Whilst these pre-1.0 releases are considered to have "Beta" status, we are committed to making them as bug-free as possible.

However, until the 1.0 release, we are aggressively reviewing and refining the functionality, organization, and interface. This is being done in an effort to make the package as coherent, intuitive, and useful as possible. To achieve this, we need help from the community of users. Specifically, we need feedback regarding all aspects of the project - everything - from which algorithms we implement, to details about our function's call signatures.

1.21.1 Python 3

Scipy 0.9.0 is the first SciPy release to support Python 3. The only module that is not yet ported is `scipy.weave`.

1.21.2 Scipy source code location to be changed

Soon after this release, Scipy will stop using SVN as the version control system, and move to Git. The development source code for Scipy can from then on be found at

> http://github.com/scipy/scipy

1.21.3 New features

Delaunay tesselations (`scipy.spatial`)

Scipy now includes routines for computing Delaunay tesselations in N dimensions, powered by the Qhull computational geometry library. Such calculations can now make use of the new `scipy.spatial.Delaunay` interface.

N-dimensional interpolation (`scipy.interpolate`)

Support for scattered data interpolation is now significantly improved. This version includes a `scipy.interpolate.griddata` function that can perform linear and nearest-neighbour interpolation for N-dimensional scattered data, in addition to cubic spline (C1-smooth) interpolation in 2D and 1D. An object-oriented interface to each interpolator type is also available.

Nonlinear equation solvers (`scipy.optimize`)

Scipy includes new routines for large-scale nonlinear equation solving in `scipy.optimize`. The following methods are implemented:

- Newton-Krylov (`scipy.optimize.newton_krylov`)
- (Generalized) secant methods:
  - Limited-memory Broyden methods (`scipy.optimize.broyden1`, `scipy.optimize.broyden2`)
  - Anderson method (`scipy.optimize.anderson`)
- Simple iterations (`scipy.optimize.diagbroyden`, `scipy.optimize.excitingmixing`, `scipy.optimize.linearmixing`)

The `scipy.optimize.nonlin` module was completely rewritten, and some of the functions were deprecated (see above).

New linear algebra routines (`scipy.linalg`)

Scipy now contains routines for effectively solving triangular equation systems (`scipy.linalg.solve_triangular`).

Improved FIR filter design functions (`scipy.signal`)

The function `scipy.signal.firwin` was enhanced to allow the design of highpass, bandpass, bandstop and multi-band FIR filters.

The function `scipy.signal.firwin2` was added. This function uses the window method to create a linear phase FIR filter with an arbitrary frequency response.

The functions `scipy.signal.kaiser_atten` and `scipy.signal.kaiser_beta` were added.

Improved statistical tests (`scipy.stats`)

A new function `scipy.stats.fisher_exact` was added, that provides Fisher's exact test for 2x2 contingency tables.

The function `scipy.stats.kendalltau` was rewritten to make it much faster (O(n log(n)) vs O(n^2)).

1.21.4 Deprecated features

Obsolete nonlinear solvers (in `scipy.optimize`)

The following nonlinear solvers from `scipy.optimize` are deprecated:

- `broyden_modified` (bad performance)
- `broyden1_modified` (bad performance)
- `broyden_generalized` (equivalent to `anderson`)
- `anderson2` (equivalent to `anderson`)
- `broyden3` (obsoleted by new limited-memory broyden methods)
- `vackar` (renamed to `diagbroyden`)

1.21.5 Removed features

The deprecated modules `helpmod`, `pexec` and `ppimport` were removed from `scipy.misc`.

The `output_type` keyword in many `scipy.ndimage` interpolation functions has been removed.

The `econ` keyword in `scipy.linalg.qr` has been removed. The same functionality is still available by specifying `mode='economic'`.

Old correlate/convolve behavior (in `scipy.signal`)

The old behavior for `scipy.signal.convolve`, `scipy.signal.convolve2d`, `scipy.signal.correlate` and `scipy.signal.correlate2d` was deprecated in 0.8.0 and has now been removed. Convolve and correlate used to swap their arguments if the second argument has dimensions larger than the first one, and the mode was relative to the input with the largest dimension. The current behavior is to never swap the inputs, which is what most people expect, and is how correlation is usually defined.

`scipy.stats`

Many functions in `scipy.stats` that are either available from numpy or have been superseded, and have been deprecated since version 0.7, have been removed: *std*, *var*, *mean*, *median*, *cov*, *corrcoef*, *z*, *zs*, *stderr*, *samplestd*, *samplevar*, *pdfapprox*, *pdf_moments* and *erfc*. These changes are mirrored in `scipy.stats.mstats`.

`scipy.sparse`

Several methods of the sparse matrix classes in `scipy.sparse` which had been deprecated since version 0.7 were removed: *save*, *rowcol*, *getdata*, *listprint*, *ensure_sorted_indices*, *matvec*, *matmat* and *rmatvec*.

The functions `spkron`, `speye`, `spidentity`, `lil_eye` and `lil_diags` were removed from `scipy.sparse`. The first three functions are still available as `scipy.sparse.kron`, `scipy.sparse.eye` and `scipy.sparse.identity`.

The *dims* and *nzmax* keywords were removed from the sparse matrix constructor. The *colind* and *rowind* attributes were removed from CSR and CSC matrices respectively.

`scipy.sparse.linalg.arpack.speigs`

A duplicated interface to the ARPACK library was removed.

1.21.6 Other changes

ARPACK interface changes

The interface to the ARPACK eigenvalue routines in `scipy.sparse.linalg` was changed for more robustness.

The eigenvalue and SVD routines now raise `ArpackNoConvergence` if the eigenvalue iteration fails to converge. If partially converged results are desired, they can be accessed as follows:

```
import numpy as np
from scipy.sparse.linalg import eigs, ArpackNoConvergence

m = np.random.randn(30, 30)
try:
    w, v = eigs(m, 6)
except ArpackNoConvergence, err:
    partially_converged_w = err.eigenvalues
    partially_converged_v = err.eigenvectors
```

Several bugs were also fixed.

The routines were moreover renamed as follows:

- eigen –> eigs
- eigen_symmetric –> eigsh
- svd –> svds

1.22 SciPy 0.8.0 Release Notes

Contents

- *SciPy 0.8.0 Release Notes*
  - *Python 3*
  - *Major documentation improvements*
  - *Deprecated features*
    - *Swapping inputs for correlation functions (scipy.signal)*
    - *Obsolete code deprecated (scipy.misc)*
    - *Additional deprecations*
  - *New features*
    - *DCT support (scipy.fftpack)*
    - *Single precision support for fft functions (scipy.fftpack)*
    - *Correlation functions now implement the usual definition (scipy.signal)*
    - *Additions and modification to LTI functions (scipy.signal)*
    - *Improved waveform generators (scipy.signal)*
    - *New functions and other changes in scipy.linalg*
    - *New function and changes in scipy.optimize*
    - *New sparse least squares solver*
    - *ARPACK-based sparse SVD*
    - *Alternative behavior available for* `scipy.constants.find`
    - *Incomplete sparse LU decompositions*
    - *Faster matlab file reader and default behavior change*
    - *Faster evaluation of orthogonal polynomials*
    - *Lambert W function*
    - *Improved hypergeometric 2F1 function*
    - *More flexible interface for Radial basis function interpolation*
  - *Removed features*
    - *scipy.io*

SciPy 0.8.0 is the culmination of 17 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.8.x branch, and on adding new features on the development trunk. This release requires Python 2.4 - 2.6 and NumPy 1.4.1 or greater.

Please note that SciPy is still considered to have "Beta" status, as we work toward a SciPy 1.0.0 release. The 1.0.0 release will mark a major milestone in the development of SciPy, after which changing the package structure or API will be much more difficult. Whilst these pre-1.0 releases are considered to have "Beta" status, we are committed to making them as bug-free as possible.

However, until the 1.0 release, we are aggressively reviewing and refining the functionality, organization, and interface.

This is being done in an effort to make the package as coherent, intuitive, and useful as possible. To achieve this, we need help from the community of users. Specifically, we need feedback regarding all aspects of the project - everything - from which algorithms we implement, to details about our function's call signatures.

1.22.1 Python 3

Python 3 compatibility is planned and is currently technically feasible, since Numpy has been ported. However, since the Python 3 compatible Numpy 1.5 has not been released yet, support for Python 3 in Scipy is not yet included in Scipy 0.8. SciPy 0.9, planned for fall 2010, will very likely include experimental support for Python 3.

1.22.2 Major documentation improvements

SciPy documentation is greatly improved.

1.22.3 Deprecated features

Swapping inputs for correlation functions (scipy.signal)

Concern correlate, correlate2d, convolve and convolve2d. If the second input is larger than the first input, the inputs are swapped before calling the underlying computation routine. This behavior is deprecated, and will be removed in scipy 0.9.0.

Obsolete code deprecated (scipy.misc)

The modules *helpmod*, *ppimport* and *pexec* from `scipy.misc` are deprecated. They will be removed from SciPy in version 0.9.

Additional deprecations

- linalg: The function *solveh_banded* currently returns a tuple containing the Cholesky factorization and the solution to the linear system. In SciPy 0.9, the return value will be just the solution.
- The function *constants.codata.find* will generate a DeprecationWarning. In Scipy version 0.8.0, the keyword argument 'disp' was added to the function, with the default value 'True'. In 0.9.0, the default will be 'False'.
- The *qshape* keyword argument of *signal.chirp* is deprecated. Use the argument *vertex_zero* instead.
- Passing the coefficients of a polynomial as the argument *f0* to *signal.chirp* is deprecated. Use the function *signal.sweep_poly* instead.
- The *io.recaster* module has been deprecated and will be removed in 0.9.0.

1.22.4 New features

DCT support (scipy.fftpack)

New realtransforms have been added, namely dct and idct for Discrete Cosine Transform; type I, II and III are available.

Single precision support for fft functions (scipy.fftpack)

fft functions can now handle single precision inputs as well: fft(x) will return a single precision array if x is single precision.

At the moment, for FFT sizes that are not composites of 2, 3, and 5, the transform is computed internally in double precision to avoid rounding error in FFTPACK.

Correlation functions now implement the usual definition (scipy.signal)

The outputs should now correspond to their matlab and R counterparts, and do what most people expect if the old_behavior=False argument is passed:

- correlate, convolve and their 2d counterparts do not swap their inputs depending on their relative shape anymore;
- correlation functions now conjugate their second argument while computing the slided sum-products, which correspond to the usual definition of correlation.

Additions and modification to LTI functions (scipy.signal)

- The functions *impulse2* and *step2* were added to `scipy.signal`. They use the function `scipy.signal.lsim2` to compute the impulse and step response of a system, respectively.
- The function `scipy.signal.lsim2` was changed to pass any additional keyword arguments to the ODE solver.

Improved waveform generators (scipy.signal)

Several improvements to the *chirp* function in `scipy.signal` were made:

- The waveform generated when *method="logarithmic"* was corrected; it now generates a waveform that is also known as an "exponential" or "geometric" chirp. (See http://en.wikipedia.org/wiki/Chirp.)
- A new *chirp* method, "hyperbolic", was added.
- Instead of the keyword *qshape*, *chirp* now uses the keyword *vertex_zero*, a boolean.
- *chirp* no longer handles an arbitrary polynomial. This functionality has been moved to a new function, *sweep_poly*.

A new function, *sweep_poly*, was added.

New functions and other changes in scipy.linalg

The functions *cho_solve_banded*, *circulant*, *companion*, *hadamard* and *leslie* were added to `scipy.linalg`.

The function *block_diag* was enhanced to accept scalar and 1D arguments, along with the usual 2D arguments.

New function and changes in scipy.optimize

The *curve_fit* function has been added; it takes a function and uses non-linear least squares to fit that to the provided data.

The *leastsq* and *fsolve* functions now return an array of size one instead of a scalar when solving for a single parameter.

New sparse least squares solver

The *lsqr* function was added to `scipy.sparse`. This routine finds a least-squares solution to a large, sparse, linear system of equations.

ARPACK-based sparse SVD

A naive implementation of SVD for sparse matrices is available in scipy.sparse.linalg.eigen.arpack. It is based on using an symmetric solver on <A, A>, and as such may not be very precise.

Alternative behavior available for `scipy.constants.find`

The keyword argument *disp* was added to the function `scipy.constants.find`, with the default value *True*. When *disp* is *True*, the behavior is the same as in Scipy version 0.7. When *False*, the function returns the list of keys instead of printing them. (In SciPy version 0.9, the default will be reversed.)

Incomplete sparse LU decompositions

Scipy now wraps SuperLU version 4.0, which supports incomplete sparse LU decompositions. These can be accessed via `scipy.sparse.linalg.spilu`. Upgrade to SuperLU 4.0 also fixes some known bugs.

Faster matlab file reader and default behavior change

We've rewritten the matlab file reader in Cython and it should now read matlab files at around the same speed that Matlab does.

The reader reads matlab named and anonymous functions, but it can't write them.

Until scipy 0.8.0 we have returned arrays of matlab structs as numpy object arrays, where the objects have attributes named for the struct fields. As of 0.8.0, we return matlab structs as numpy structured arrays. You can get the older behavior by using the optional `struct_as_record=False` keyword argument to `scipy.io.loadmat` and friends.

There is an inconsistency in the matlab file writer, in that it writes numpy 1D arrays as column vectors in matlab 5 files, and row vectors in matlab 4 files. We will change this in the next version, so both write row vectors. There is a *FutureWarning* when calling the writer to warn of this change; for now we suggest using the `oned_as='row'` keyword argument to `scipy.io.savemat` and friends.

Faster evaluation of orthogonal polynomials

Values of orthogonal polynomials can be evaluated with new vectorized functions in `scipy.special`: *eval_legendre*, *eval_chebyt*, *eval_chebyu*, *eval_chebyc*, *eval_chebys*, *eval_jacobi*, *eval_laguerre*, *eval_genlaguerre*, *eval_hermite*, *eval_hermitenorm*, *eval_gegenbauer*, *eval_sh_legendre*, *eval_sh_chebyt*, *eval_sh_chebyu*, *eval_sh_jacobi*. This is faster than constructing the full coefficient representation of the polynomials, which was previously the only available way.

Note that the previous orthogonal polynomial routines will now also invoke this feature, when possible.

Lambert W function

`scipy.special.lambertw` can now be used for evaluating the Lambert W function.

Improved hypergeometric 2F1 function

Implementation of `scipy.special.hyp2f1` for real parameters was revised. The new version should produce accurate values for all real parameters.

More flexible interface for Radial basis function interpolation

The `scipy.interpolate.Rbf` class now accepts a callable as input for the "function" argument, in addition to the built-in radial basis functions which can be selected with a string argument.

1.22.5 Removed features

scipy.stsci: the package was removed

The module *scipy.misc.limits* was removed.

The IO code in both NumPy and SciPy is being extensively reworked. NumPy will be where basic code for reading and writing NumPy arrays is located, while SciPy will house file readers and writers for various data formats (data, audio, video, images, matlab, etc.).

Several functions in `scipy.io` are removed in the 0.8.0 release including: *npfile*, *save*, *load*, *create_module*, *create_shelf*, *objload*, *objsave*, *fopen*, *read_array*, *write_array*, *fread*, *fwrite*, *bswap*, *packbits*, *unpackbits*, and *convert_objectarray*. Some of these functions have been replaced by NumPy's raw reading and writing capabilities, memory-mapping capabilities, or array methods. Others have been moved from SciPy to NumPy, since basic array reading and writing capability is now handled by NumPy.

1.23 SciPy 0.7.2 Release Notes

Contents

- *SciPy 0.7.2 Release Notes*

SciPy 0.7.2 is a bug-fix release with no new features compared to 0.7.1. The only change is that all C sources from Cython code have been regenerated with Cython 0.12.1. This fixes the incompatibility between binaries of SciPy 0.7.1 and NumPy 1.4.

1.24 SciPy 0.7.1 Release Notes

Contents

- *SciPy 0.7.1 Release Notes*
  - *scipy.io*
  - *scipy.odr*
  - *scipy.signal*
  - *scipy.sparse*

- *scipy.special*
- *scipy.stats*
- *Windows binaries for python 2.6*
- *Universal build for scipy*

SciPy 0.7.1 is a bug-fix release with no new features compared to 0.7.0.

Bugs fixed:

- Several fixes in Matlab file IO

Bugs fixed:

- Work around a failure with Python 2.6

Memory leak in lfilter have been fixed, as well as support for array object

Bugs fixed:

- #880, #925: lfilter fixes
- #871: bicgstab fails on Win32

Bugs fixed:

- #883: scipy.io.mmread with scipy.sparse.lil_matrix broken
- lil_matrix and csc_matrix reject now unexpected sequences, cf. http://thread.gmane.org/gmane.comp.python.scientific.user/19996

Several bugs of varying severity were fixed in the special functions:

- #503, #640: iv: problems at large arguments fixed by new implementation
- #623: jv: fix errors at large arguments
- #679: struve: fix wrong output for v < 0
- #803: pbdv produces invalid output
- #804: lqmn: fix crashes on some input
- #823: betainc: fix documentation
- #834: exp1 strange behavior near negative integer values
- #852: jn_zeros: more accurate results for large s, also in jnp/yn/ynp_zeros
- #853: jv, yv, iv: invalid results for non-integer v < 0, complex x
- #854: jv, yv, iv, kv: return nan more consistently when out-of-domain
- #927: ellipj: fix segfault on Windows
- #946: ellpj: fix segfault on Mac OS X/python 2.6 combination.
- ive, jve, yve, kv, kve: with real-valued input, return nan for out-of-domain instead of returning only the real part of the result.

Also, when `scipy.special.errprint(1)` has been enabled, warning messages are now issued as Python warnings instead of printing them to stderr.

- linregress, mannwhitneyu, describe: errors fixed
- kstwobign, norm, expon, exponweib, exponpow, frechet, genexpon, rdist, truncexpon, planck: improvements to numerical accuracy in distributions

1.24.1 Windows binaries for python 2.6

python 2.6 binaries for windows are now included. The binary for python 2.5 requires numpy 1.2.0 or above, and the one for python 2.6 requires numpy 1.3.0 or above.

1.24.2 Universal build for scipy

Mac OS X binary installer is now a proper universal build, and does not depend on gfortran anymore (libgfortran is statically linked). The python 2.5 version of scipy requires numpy 1.2.0 or above, the python 2.6 version requires numpy 1.3.0 or above.

1.25 SciPy 0.7.0 Release Notes

Contents

- *SciPy 0.7.0 Release Notes*
  - *Python 2.6 and 3.0*
  - *Major documentation improvements*
  - *Running Tests*
  - *Building SciPy*
  - *Sandbox Removed*
  - *Sparse Matrices*
  - *Statistics package*
  - *Reworking of IO package*
  - *New Hierarchical Clustering module*
  - *New Spatial package*
  - *Reworked fftpack package*
  - *New Constants package*
  - *New Radial Basis Function module*
  - *New complex ODE integrator*
  - *New generalized symmetric and hermitian eigenvalue problem solver*
  - *Bug fixes in the interpolation package*
  - *Weave clean up*
  - *Known problems*

SciPy 0.7.0 is the culmination of 16 months of hard work. It contains many new features, numerous bug-fixes, improved test coverage and better documentation. There have been a number of deprecations and API changes in this release, which are documented below. All users are encouraged to upgrade to this release, as there are a large number of bug-fixes and optimizations. Moreover, our development attention will now shift to bug-fix releases on the 0.7.x branch, and on adding new features on the development trunk. This release requires Python 2.4 or 2.5 and NumPy 1.2 or greater.

Please note that SciPy is still considered to have "Beta" status, as we work toward a SciPy 1.0.0 release. The 1.0.0 release will mark a major milestone in the development of SciPy, after which changing the package structure or API will be much more difficult. Whilst these pre-1.0 releases are considered to have "Beta" status, we are committed to making them as bug-free as possible. For example, in addition to fixing numerous bugs in this release, we have also doubled the number of unit tests since the last release.

However, until the 1.0 release, we are aggressively reviewing and refining the functionality, organization, and interface. This is being done in an effort to make the package as coherent, intuitive, and useful as possible. To achieve this, we need help from the community of users. Specifically, we need feedback regarding all aspects of the project - everything - from which algorithms we implement, to details about our function's call signatures.

Over the last year, we have seen a rapid increase in community involvement, and numerous infrastructure improvements to lower the barrier to contributions (e.g., more explicit coding standards, improved testing infrastructure, better documentation tools). Over the next year, we hope to see this trend continue and invite everyone to become more involved.

1.25.1 Python 2.6 and 3.0

A significant amount of work has gone into making SciPy compatible with Python 2.6; however, there are still some issues in this regard. The main issue with 2.6 support is NumPy. On UNIX (including Mac OS X), NumPy 1.2.1 mostly works, with a few caveats. On Windows, there are problems related to the compilation process. The upcoming NumPy 1.3 release will fix these problems. Any remaining issues with 2.6 support for SciPy 0.7 will be addressed in a bug-fix release.

Python 3.0 is not supported at all; it requires NumPy to be ported to Python 3.0. This requires immense effort, since a lot of C code has to be ported. The transition to 3.0 is still under consideration; currently, we don't have any timeline or roadmap for this transition.

1.25.2 Major documentation improvements

SciPy documentation is greatly improved; you can view a HTML reference manual online or download it as a PDF file. The new reference guide was built using the popular Sphinx tool.

This release also includes an updated tutorial, which hadn't been available since SciPy was ported to NumPy in 2005. Though not comprehensive, the tutorial shows how to use several essential parts of Scipy. It also includes the `ndimage` documentation from the `numarray` manual.

Nevertheless, more effort is needed on the documentation front. Luckily, contributing to Scipy documentation is now easier than before: if you find that a part of it requires improvements, and want to help us out, please register a user name in our web-based documentation editor at https://docs.scipy.org/ and correct the issues.

1.25.3 Running Tests

NumPy 1.2 introduced a new testing framework based on nose. Starting with this release, SciPy now uses the new NumPy test framework as well. Taking advantage of the new testing framework requires `nose` version 0.10, or later. One major advantage of the new framework is that it greatly simplifies writing unit tests - which has all ready paid off, given the rapid increase in tests. To run the full test suite:

```
>>> import scipy
>>> scipy.test('full')
```

For more information, please see The NumPy/SciPy Testing Guide.

We have also greatly improved our test coverage. There were just over 2,000 unit tests in the 0.6.0 release; this release nearly doubles that number, with just over 4,000 unit tests.

1.25.4 Building SciPy

Support for NumScons has been added. NumScons is a tentative new build system for NumPy/SciPy, using SCons at its core.

SCons is a next-generation build system, intended to replace the venerable `Make` with the integrated functionality of `autoconf`/`automake` and `ccache`. Scons is written in Python and its configuration files are Python scripts. NumScons is meant to replace NumPy's custom version of `distutils` providing more advanced functionality, such as `autoconf`, improved fortran support, more tools, and support for `numpy.distutils`/`scons` cooperation.

1.25.5 Sandbox Removed

While porting SciPy to NumPy in 2005, several packages and modules were moved into `scipy.sandbox`. The sandbox was a staging ground for packages that were undergoing rapid development and whose APIs were in flux. It was also a place where broken code could live. The sandbox has served its purpose well, but was starting to create confusion. Thus `scipy.sandbox` was removed. Most of the code was moved into `scipy`, some code was made into a `scikit`, and the remaining code was just deleted, as the functionality had been replaced by other code.

1.25.6 Sparse Matrices

Sparse matrices have seen extensive improvements. There is now support for integer dtypes such `int8`, `uint32`, etc. Two new sparse formats were added:

- new class `dia_matrix` : the sparse DIAgonal format
- new class `bsr_matrix` : the Block CSR format

Several new sparse matrix construction functions were added:

- `sparse.kron` : sparse Kronecker product
- `sparse.bmat` : sparse version of `numpy.bmat`
- `sparse.vstack` : sparse version of `numpy.vstack`
- `sparse.hstack` : sparse version of `numpy.hstack`

Extraction of submatrices and nonzero values have been added:

- `sparse.tril` : extract lower triangle
- `sparse.triu` : extract upper triangle
- `sparse.find` : nonzero values and their indices

`csr_matrix` and `csc_matrix` now support slicing and fancy indexing (e.g., `A[1:3, 4:7]` and `A[[3,2,6,8],:]`). Conversions among all sparse formats are now possible:

- using member functions such as `.tocsr()` and `.tolil()`
- using the `.asformat()` member function, e.g. `A.asformat('csr')`
- using constructors `A = lil_matrix([[1,2]]); B = csr_matrix(A)`

All sparse constructors now accept dense matrices and lists of lists. For example:

- `A = csr_matrix( rand(3,3) )` and `B = lil_matrix( [[1,2],[3,4]] )`

The handling of diagonals in the `spdiags` function has been changed. It now agrees with the MATLAB(TM) function of the same name.

Numerous efficiency improvements to format conversions and sparse matrix arithmetic have been made. Finally, this release contains numerous bugfixes.

1.25.7 Statistics package

Statistical functions for masked arrays have been added, and are accessible through `scipy.stats.mstats`. The functions are similar to their counterparts in `scipy.stats` but they have not yet been verified for identical interfaces and algorithms.

Several bugs were fixed for statistical functions, of those, `kstest` and `percentileofscore` gained new keyword arguments.

Added deprecation warning for `mean`, `median`, `var`, `std`, `cov`, and `corrcoef`. These functions should be replaced by their numpy counterparts. Note, however, that some of the default options differ between the `scipy.stats` and numpy versions of these functions.

Numerous bug fixes to `stats.distributions`: all generic methods now work correctly, several methods in individual distributions were corrected. However, a few issues remain with higher moments (`skew`, `kurtosis`) and entropy. The maximum likelihood estimator, `fit`, does not work out-of-the-box for some distributions - in some cases, starting values have to be carefully chosen, in other cases, the generic implementation of the maximum likelihood method might not be the numerically appropriate estimation method.

We expect more bugfixes, increases in numerical precision and enhancements in the next release of scipy.

1.25.8 Reworking of IO package

The IO code in both NumPy and SciPy is being extensively reworked. NumPy will be where basic code for reading and writing NumPy arrays is located, while SciPy will house file readers and writers for various data formats (data, audio, video, images, matlab, etc.).

Several functions in `scipy.io` have been deprecated and will be removed in the 0.8.0 release including `npfile`, `save`, `load`, `create_module`, `create_shelf`, `objload`, `objsave`, `fopen`, `read_array`, `write_array`, `fread`, `fwrite`, `bswap`, `packbits`, `unpackbits`, and `convert_objectarray`. Some of these functions have been replaced by NumPy's raw reading and writing capabilities, memory-mapping capabilities, or array methods. Others have been moved from SciPy to NumPy, since basic array reading and writing capability is now handled by NumPy.

The Matlab (TM) file readers/writers have a number of improvements:

- default version 5
- v5 writers for structures, cell arrays, and objects
- v5 readers/writers for function handles and 64-bit integers
- new struct_as_record keyword argument to `loadmat`, which loads struct arrays in matlab as record arrays in numpy
- string arrays have `dtype='U...'` instead of `dtype=object`
- `loadmat` no longer squeezes singleton dimensions, i.e. `squeeze_me=False` by default

1.25.9 New Hierarchical Clustering module

This module adds new hierarchical clustering functionality to the `scipy.cluster` package. The function interfaces are similar to the functions provided MATLAB(TM)'s Statistics Toolbox to help facilitate easier migration to

the NumPy/SciPy framework. Linkage methods implemented include single, complete, average, weighted, centroid, median, and ward.

In addition, several functions are provided for computing inconsistency statistics, cophenetic distance, and maximum distance between descendants. The `fcluster` and `fclusterdata` functions transform a hierarchical clustering into a set of flat clusters. Since these flat clusters are generated by cutting the tree into a forest of trees, the `leaders` function takes a linkage and a flat clustering, and finds the root of each tree in the forest. The `ClusterNode` class represents a hierarchical clusterings as a field-navigable tree object. `to_tree` converts a matrix-encoded hierarchical clustering to a `ClusterNode` object. Routines for converting between MATLAB and SciPy linkage encodings are provided. Finally, a `dendrogram` function plots hierarchical clusterings as a dendrogram, using matplotlib.

1.25.10 New Spatial package

The new spatial package contains a collection of spatial algorithms and data structures, useful for spatial statistics and clustering applications. It includes rapidly compiled code for computing exact and approximate nearest neighbors, as well as a pure-python kd-tree with the same interface, but that supports annotation and a variety of other algorithms. The API for both modules may change somewhat, as user requirements become clearer.

It also includes a `distance` module, containing a collection of distance and dissimilarity functions for computing distances between vectors, which is useful for spatial statistics, clustering, and kd-trees. Distance and dissimilarity functions provided include Bray-Curtis, Canberra, Chebyshev, City Block, Cosine, Dice, Euclidean, Hamming, Jaccard, Kulsinski, Mahalanobis, Matching, Minkowski, Rogers-Tanimoto, Russell-Rao, Squared Euclidean, Standardized Euclidean, Sokal-Michener, Sokal-Sneath, and Yule.

The `pdist` function computes pairwise distance between all unordered pairs of vectors in a set of vectors. The `cdist` computes the distance on all pairs of vectors in the Cartesian product of two sets of vectors. Pairwise distance matrices are stored in condensed form; only the upper triangular is stored. `squareform` converts distance matrices between square and condensed forms.

1.25.11 Reworked fftpack package

FFTW2, FFTW3, MKL and DJBFFT wrappers have been removed. Only (NETLIB) fftpack remains. By focusing on one backend, we hope to add new features - like float32 support - more easily.

1.25.12 New Constants package

`scipy.constants` provides a collection of physical constants and conversion factors. These constants are taken from CODATA Recommended Values of the Fundamental Physical Constants: 2002. They may be found at physics.nist.gov/constants. The values are stored in the dictionary physical_constants as a tuple containing the value, the units, and the relative precision - in that order. All constants are in SI units, unless otherwise stated. Several helper functions are provided.

1.25.13 New Radial Basis Function module

`scipy.interpolate` now contains a Radial Basis Function module. Radial basis functions can be used for smoothing/interpolating scattered data in n-dimensions, but should be used with caution for extrapolation outside of the observed data range.

1.25.14 New complex ODE integrator

`scipy.integrate.ode` now contains a wrapper for the ZVODE complex-valued ordinary differential equation solver (by Peter N. Brown, Alan C. Hindmarsh, and George D. Byrne).

1.25.15 New generalized symmetric and hermitian eigenvalue problem solver

`scipy.linalg.eigh` now contains wrappers for more LAPACK symmetric and hermitian eigenvalue problem solvers. Users can now solve generalized problems, select a range of eigenvalues only, and choose to use a faster algorithm at the expense of increased memory usage. The signature of the `scipy.linalg.eigh` changed accordingly.

1.25.16 Bug fixes in the interpolation package

The shape of return values from `scipy.interpolate.interp1d` used to be incorrect, if interpolated data had more than 2 dimensions and the axis keyword was set to a non-default value. This has been fixed. Moreover, `interp1d` returns now a scalar (0D-array) if the input is a scalar. Users of `scipy.interpolate.interp1d` may need to revise their code if it relies on the previous behavior.

1.25.17 Weave clean up

There were numerous improvements to `scipy.weave`. `blitz++` was relicensed by the author to be compatible with the SciPy license. `wx_spec.py` was removed.

1.25.18 Known problems

Here are known problems with scipy 0.7.0:

- weave test failures on windows: those are known, and are being revised.
- weave test failure with gcc 4.3 (std::labs): this is a gcc 4.3 bug. A workaround is to add #include <cstdlib> in scipy/weave/blitz/blitz/funcs.h (line 27). You can make the change in the installed scipy (in site-packages).

CHAPTER

TWO

API - IMPORTING FROM SCIPY

In Python the distinction between what is the public API of a library and what are private implementation details is not always clear. Unlike in other languages like Java, it is possible in Python to access "private" function or objects. Occasionally this may be convenient, but be aware that if you do so your code may break without warning in future releases. Some widely understood rules for what is and isn't public in Python are:

- Methods / functions / classes and module attributes whose names begin with a leading underscore are private.
- If a class name begins with a leading underscore none of its members are public, whether or not they begin with a leading underscore.
- If a module name in a package begins with a leading underscore none of its members are public, whether or not they begin with a leading underscore.
- If a module or package defines `__all__` that authoritatively defines the public interface.
- If a module or package doesn't define `__all__` then all names that don't start with a leading underscore are public.

Note: Reading the above guidelines one could draw the conclusion that every private module or object starts with an underscore. This is not the case; the presence of underscores do mark something as private, but the absence of underscores do not mark something as public.

In Scipy there are modules whose names don't start with an underscore, but that should be considered private. To clarify which modules these are we define below what the public API is for Scipy, and give some recommendations for how to import modules/functions/objects from Scipy.

2.1 Guidelines for importing functions from Scipy

The scipy namespace itself only contains functions imported from numpy. These functions still exist for backwards compatibility, but should be imported from numpy directly.

Everything in the namespaces of scipy submodules is public. In general, it is recommended to import functions from submodule namespaces. For example, the function `curve_fit` (defined in scipy/optimize/minpack.py) should be imported like this:

```
from scipy import optimize
result = optimize.curve_fit(...)
```

This form of importing submodules is preferred for all submodules except `scipy.io` (because `io` is also the name of a module in the Python stdlib):

```
from scipy import interpolate
from scipy import integrate
import scipy.io as spio
```

In some cases, the public API is one level deeper. For example the `scipy.sparse.linalg` module is public, and the functions it contains are not available in the `scipy.sparse` namespace. Sometimes it may result in more easily understandable code if functions are imported from one level deeper. For example, in the following it is immediately clear that `lomax` is a distribution if the second form is chosen:

```
# first form
from scipy import stats
stats.lomax(...)

# second form
from scipy.stats import distributions
distributions.lomax(...)
```

In that case the second form can be chosen, **if** it is documented in the next section that the submodule in question is public.

2.2 API definition

Every submodule listed below is public. That means that these submodules are unlikely to be renamed or changed in an incompatible way, and if that is necessary a deprecation warning will be raised for one Scipy release before the change is made.

- scipy.cluster
  - vq
  - hierarchy
- scipy.constants
- scipy.fftpack
- scipy.integrate
- scipy.interpolate
- scipy.io
  - arff
  - harwell_boeing
  - idl
  - matlab
  - netcdf
  - wavfile
- scipy.linalg
  - scipy.linalg.blas
  - scipy.linalg.cython_blas
  - scipy.linalg.lapack

- scipy.linalg.cython_lapack
- scipy.linalg.interpolative
- scipy.misc
- scipy.ndimage
- scipy.odr
- scipy.optimize
- scipy.signal
- scipy.sparse
  - linalg
  - csgraph
- scipy.spatial
  - distance
- scipy.special
- scipy.stats
  - distributions
  - mstats

CHAPTER

THREE

TUTORIAL

Tutorials with worked examples and background information for most SciPy submodules.

3.1 SciPy Tutorial

3.1.1 Introduction

Contents

- *Introduction*
  - *SciPy Organization*
  - *Finding Documentation*

SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems such as MATLAB, IDL, Octave, R-Lab, and SciLab.

The additional benefit of basing SciPy on Python is that this also makes a powerful programming language available for use in developing sophisticated programs and specialized applications. Scientific applications using SciPy benefit from the development of additional modules in numerous niches of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This tutorial will acquaint the first-time user of SciPy with some of its most important features. It assumes that the user has already installed the SciPy package. Some general Python facility is also assumed, such as could be acquired by working through the Python distribution's Tutorial. For further introductory help the user is directed to the Numpy documentation.

For brevity and convenience, we will often assume that the main packages (numpy, scipy, and matplotlib) have been imported as:

```
>>> import numpy as np
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

These are the import conventions that our community has adopted after discussion on public mailing lists. You will see these conventions used throughout NumPy and SciPy source code and documentation. While we obviously don't require you to follow these conventions in your own code, it is highly recommended.

SciPy Organization

SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

Scipy sub-packages need to be imported separately, for example:

```
>>> from scipy import linalg, optimize
```

Because of their ubiquitousness, some of the functions in these subpackages are also made available in the `scipy` namespace to ease their use in interactive sessions and programs. In addition, many basic array functions from `numpy` are also available at the top-level of the `scipy` package. Before looking at the sub-packages individually, we will first look at some of these common functions.

Finding Documentation

SciPy and NumPy have documentation versions in both HTML and PDF format available at https://docs.scipy.org/, that cover nearly all available functionality. However, this documentation is still work-in-progress and some parts may be incomplete or sparse. As we are a volunteer organization and depend on the community for growth, your participation - everything from providing feedback to improving the documentation and code - is welcome and actively encouraged.

Python's documentation strings are used in SciPy for on-line documentation. There are two methods for reading them and getting help. One is Python's command `help` in the `pydoc` module. Entering this command with no arguments (i.e. `>>> help`) launches an interactive help session that allows searching through the keywords and modules available to all of Python. Secondly, running the command *help(obj)* with an object as the argument displays that object's calling signature, and documentation string.

The pydoc method of `help` is sophisticated but uses a pager to display the text. Sometimes this can interfere with the terminal you are running the interactive session within. A numpy/scipy-specific help system is also available under the command `numpy.info`. The signature and documentation string for the object passed to the `help` command are printed to standard output (or to a writeable object passed as the third argument). The second keyword argument of `numpy.info` defines the maximum width of the line for printing. If a module is passed as the argument to `help` then a list of the functions and classes defined in that module is printed. For example:

```
>>> np.info(optimize.fmin)
 fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None,
      full_output=0, disp=1, retall=0, callback=None)
```

```
Minimize a function using the downhill simplex algorithm.

Parameters
----------
func : callable func(x,*args)
    The objective function to be minimized.
x0 : ndarray
    Initial guess.
args : tuple
    Extra arguments passed to func, i.e. ``f(x,*args)``.
callback : callable
    Called after each iteration, as callback(xk), where xk is the
    current parameter vector.

Returns
-------
xopt : ndarray
    Parameter that minimizes function.
fopt : float
    Value of function at minimum: ``fopt = func(xopt)``.
iter : int
    Number of iterations performed.
funcalls : int
    Number of function calls made.
warnflag : int
    1 : Maximum number of function evaluations made.
    2 : Maximum number of iterations reached.
allvecs : list
    Solution at each iteration.

Other parameters
----------------
xtol : float
    Relative error in xopt acceptable for convergence.
ftol : number
    Relative error in func(xopt) acceptable for convergence.
maxiter : int
    Maximum number of iterations to perform.
maxfun : number
    Maximum number of function evaluations to make.
full_output : bool
    Set to True if fopt and warnflag outputs are desired.
disp : bool
    Set to True to print convergence messages.
retall : bool
    Set to True to return list of solutions at each iteration.

Notes
-----
Uses a Nelder-Mead simplex algorithm to find the minimum of function of
one or more variables.
```

Another useful command is `source`. When given a function written in Python as an argument, it prints out a listing of the source code for that function. This can be helpful in learning about an algorithm or understanding exactly what a function is doing with its arguments. Also don't forget about the Python command `dir` which can be used to look at the namespace of a module or package.

3.1.2 Basic functions

Contents

- *Basic functions*
  - *Interaction with Numpy*
    - *Index Tricks*
    - *Shape manipulation*
    - *Polynomials*
    - *Vectorizing functions (vectorize)*
    - *Type handling*
    - *Other useful functions*

Interaction with Numpy

Scipy builds on Numpy, and for all basic array handling needs you can use Numpy functions:

```
>>> import numpy as np
>>> np.some_function()
```

Rather than giving a detailed description of each of these functions (which is available in the Numpy Reference Guide or by using the `help`, `info` and `source` commands), this tutorial will discuss some of the more useful commands which require a little introduction to use to their full potential.

To use functions from some of the Scipy modules, you can do:

```
>>> from scipy import some_module
>>> some_module.some_function()
```

The top level of `scipy` also contains functions from `numpy` and `numpy.lib.scimath`. However, it is better to use them directly from the `numpy` module instead.

Index Tricks

There are some class instances that make special use of the slicing functionality to provide efficient means for array construction. This part will discuss the operation of `np.mgrid` , `np.ogrid` , `np.r_` , and `np.c_` for quickly constructing arrays.

For example, rather than writing something like the following

```
>>> a = np.concatenate(([3], [0]*5, np.arange(-1, 1.002, 2/9.0)))
```

with the `r_` command one can enter this as

```
>>> a = np.r_[3,[0]*5,-1:1:10j]
```

which can ease typing and make for more readable code. Notice how objects are concatenated, and the slicing syntax is (ab)used to construct ranges. The other term that deserves a little explanation is the use of the complex number 10j as the step size in the slicing syntax. This non-standard use allows the number to be interpreted as the number of points to produce in the range rather than as a step size (note we would have used the long integer notation, 10L, but this notation may go away in Python as the integers become unified). This non-standard usage may be unsightly to

some, but it gives the user the ability to quickly construct complicated vectors in a very readable fashion. When the number of points is specified in this way, the end- point is inclusive.

The "r" stands for row concatenation because if the objects between commas are 2 dimensional arrays, they are stacked by rows (and thus must have commensurate columns). There is an equivalent command `c_` that stacks 2d arrays by columns but works identically to `r_` for 1d arrays.

Another very useful class instance which makes use of extended slicing notation is the function `mgrid`. In the simplest case, this function can be used to construct 1d ranges as a convenient substitute for arange. It also allows the use of complex-numbers in the step-size to indicate the number of points to place between the (inclusive) end-points. The real purpose of this function however is to produce N, N-d arrays which provide coordinate arrays for an N-dimensional volume. The easiest way to understand this is with an example of its usage:

```
>>> np.mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> np.mgrid[0:5:4j,0:5:4j]
array([[[ 0.    ,  0.    ,  0.    ,  0.    ],
        [ 1.6667,  1.6667,  1.6667,  1.6667],
        [ 3.3333,  3.3333,  3.3333,  3.3333],
        [ 5.    ,  5.    ,  5.    ,  5.    ]],
       [[ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ]]])
```

Having meshed arrays like this is sometimes very useful. However, it is not always needed just to evaluate some N-dimensional function over a grid due to the array-broadcasting rules of Numpy and SciPy. If this is the only purpose for generating a meshgrid, you should instead use the function `ogrid` which generates an "open" grid using `newaxis` judiciously to create N, N-d arrays where only one dimension in each array has length greater than 1. This will save memory and create the same result if the only purpose for the meshgrid is to generate sample points for evaluation of an N-d function.

Shape manipulation

In this category of functions are routines for squeezing out length- one dimensions from N-dimensional arrays, ensuring that an array is at least 1-, 2-, or 3-dimensional, and stacking (concatenating) arrays by rows, columns, and "pages "(in the third dimension). Routines for splitting arrays (roughly the opposite of stacking arrays) are also available.

Polynomials

There are two (interchangeable) ways to deal with 1-d polynomials in SciPy. The first is to use the `poly1d` class from Numpy. This class accepts coefficients or polynomial roots to initialize a polynomial. The polynomial object can then be manipulated in algebraic expressions, integrated, differentiated, and evaluated. It even prints like a polynomial:

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5])
>>> print p
   2
3 x + 4 x + 5
>>> print p*p
```

```
   4      3      2
9 x + 24 x + 46 x + 40 x + 25
>>> print p.integ(k=6)
   3     2
1 x + 2 x + 5 x + 6
>>> print p.deriv()
6 x + 4
>>> p([4, 5])
array([ 69, 100])
```

The other way to handle polynomials is as an array of coefficients with the first element of the array giving the coefficient of the highest power. There are explicit functions to add, subtract, multiply, divide, integrate, differentiate, and evaluate polynomials represented as sequences of coefficients.

Vectorizing functions (vectorize)

One of the features that NumPy provides is a class `vectorize` to convert an ordinary Python function which accepts scalars and returns scalars into a "vectorized-function" with the same broadcasting rules as other Numpy functions (*i.e.* the Universal functions, or ufuncs). For example, suppose you have a Python function named `addsubtract` defined as:

```
>>> def addsubtract(a,b):
...     if a > b:
...         return a - b
...     else:
...         return a + b
```

which defines a function of two scalar variables and returns a scalar result. The class vectorize can be used to "vectorize "this function so that

```
>>> vec_addsubtract = np.vectorize(addsubtract)
```

returns a function which takes array arguments and returns an array result:

```
>>> vec_addsubtract([0,3,6,9],[1,3,5,7])
array([1, 6, 1, 2])
```

This particular function could have been written in vector form without the use of `vectorize` . But, what if the function you have written is the result of some optimization or integration routine. Such functions can likely only be vectorized using `vectorize.`

Type handling

Note the difference between `np.iscomplex`/`np.isreal` and `np.iscomplexobj`/`np.isrealobj`. The former command is array based and returns byte arrays of ones and zeros providing the result of the element-wise test. The latter command is object based and returns a scalar describing the result of the test on the entire object.

Often it is required to get just the real and/or imaginary part of a complex number. While complex numbers and arrays have attributes that return those values, if one is not sure whether or not the object will be complex-valued, it is better to use the functional forms `np.real` and `np.imag` . These functions succeed for anything that can be turned into a Numpy array. Consider also the function `np.real_if_close` which transforms a complex-valued number with tiny imaginary part into a real number.

Occasionally the need to check whether or not a number is a scalar (Python (long)int, Python float, Python complex, or rank-0 array) occurs in coding. This functionality is provided in the convenient function `np.isscalar` which returns a 1 or a 0.

Finally, ensuring that objects are a certain Numpy type occurs often enough that it has been given a convenient interface in SciPy through the use of the `np.cast` dictionary. The dictionary is keyed by the type it is desired to cast to and the dictionary stores functions to perform the casting. Thus, `np.cast['f'](d)` returns an array of `np.float32` from *d*. This function is also useful as an easy way to get a scalar of a certain type:

```
>>> np.cast['f'](np.pi)
array(3.1415927410125732, dtype=float32)
```

Other useful functions

There are also several other useful functions which should be mentioned. For doing phase processing, the functions `angle`, and `unwrap` are useful. Also, the `linspace` and `logspace` functions return equally spaced samples in a linear or log scale. Finally, it's useful to be aware of the indexing capabilities of Numpy. Mention should be made of the function `select` which extends the functionality of `where` to include multiple conditions and multiple choices. The calling convention is `select(condlist,choicelist,default=0)`. `select` is a vectorized form of the multiple if-statement. It allows rapid construction of a function which returns an array of results based on a list of conditions. Each element of the return array is taken from the array in a `choicelist` corresponding to the first condition in `condlist` that is true. For example

```
>>> x = np.r_[-2:3]
>>> x
array([-2, -1,  0,  1,  2])
>>> np.select([x > 3, x >= 0], [0, x+2])
array([0, 0, 2, 3, 4])
```

Some additional useful functions can also be found in the module *scipy.misc*. For example the `factorial` and `comb` functions compute $n!$ and $n!/k!(n-k)!$ using either exact integer arithmetic (thanks to Python's Long integer object), or by using floating-point precision and the gamma function. Another function returns a common image used in image processing: `lena`.

Finally, two functions are provided that are useful for approximating derivatives of functions using discrete-differences. The function `central_diff_weights` returns weighting coefficients for an equally-spaced N-point approximation to the derivative of order o. These weights must be multiplied by the function corresponding to these points and the results added to obtain the derivative approximation. This function is intended for use when only samples of the function are available. When the function is an object that can be handed to a routine and evaluated, the function `derivative` can be used to automatically evaluate the object at the correct points to obtain an N-point approximation to the o-th derivative at a given point.

3.1.3 Special functions (scipy.special)

The main feature of the *scipy.special* package is the definition of numerous special functions of mathematical physics. Available functions include airy, elliptic, bessel, gamma, beta, hypergeometric, parabolic cylinder, mathieu, spheroidal wave, struve, and kelvin. There are also some low-level stats functions that are not intended for general use as an easier interface to these functions is provided by the `stats` module. Most of these functions can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. Many of these functions also accept complex numbers as input. For a complete list of the available functions with a one-line description type `>>> help(special)`. Each function also has its own documentation accessible using help. If you don't see a function you need, consider writing it and contributing it to the library. You can write the function in either C, Fortran, or Python. Look in the source code of the library for examples of each of these kinds of functions.

Bessel functions of real order(`jn`, `jn_zeros`)

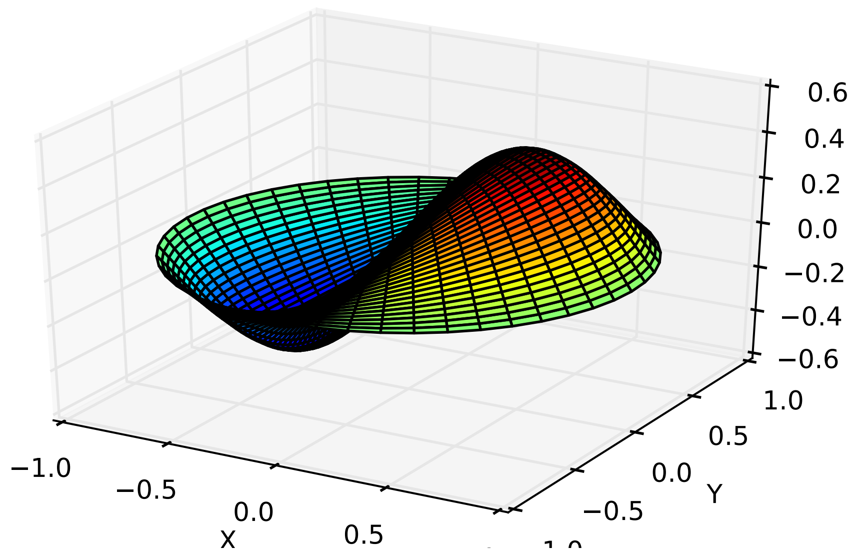
Bessel functions are a family of solutions to Bessel's differential equation with real or complex order alpha:

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - \alpha^2) y = 0$$

Among other uses, these functions arise in wave propagation problems such as the vibrational modes of a thin drum head. Here is an example of a circular drum head anchored at the edge:

```
>>> from scipy import special
>>> def drumhead_height(n, k, distance, angle, t):
...    kth_zero = special.jn_zeros(n, k)[-1]
...    return np.cos(t) * np.cos(n*angle) * special.jn(n, distance*kth_zero)
>>> theta = np.r_[0:2*np.pi:50j]
>>> radius = np.r_[0:1:50j]
>>> x = np.array([r * np.cos(theta) for r in radius])
>>> y = np.array([r * np.sin(theta) for r in radius])
>>> z = np.array([drumhead_height(1, 1, r, theta, 0.5) for r in radius])
```

```
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from matplotlib import cm
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.jet)
>>> ax.set_xlabel('X')
>>> ax.set_ylabel('Y')
>>> ax.set_zlabel('Z')
>>> plt.show()
```



Cython Bindings for Special Functions (`scipy.special.cython_special`)

Scipy also offers Cython bindings for scalar, typed versions of many of the functions in special. The following Cython code gives a simple example of how to use these functions:

```
cimport scipy.special.cython_special as csc

cdef:
    double x = 1
    double complex z = 1 + 1j
    double si, ci, rgam
    double complex cgam

rgam = csc.gamma(x)
print(rgam)
cgam = csc.gamma(z)
print(cgam)
csc.sici(x, &si, &ci)
print(si, ci)
```

(See the Cython documentation for help with compiling Cython.) In the example the function `csc.gamma` works essentially like its ufunc counterpart *gamma*, though it takes C types as arguments instead of NumPy arrays. Note in particular that the function is overloaded to support real and complex arguments; the correct variant is selected at compile time. The function `csc.sici` works slightly differently from *sici*; for the ufunc we could write `ai, bi = sici(x)` whereas in the Cython version multiple return values are passed as pointers. It might help to think of this as analogous to calling a ufunc with an output array: `sici(x, out=(si, ci))`.

There are two potential advantages to using the Cython bindings:

- They avoid Python function overhead
- They do not require the Python Global Interpreter Lock (GIL)

The following sections discuss how to use these advantages to potentially speed up your code, though of course one should always profile the code first to make sure putting in the extra effort will be worth it.

Avoiding Python Function Overhead

For the ufuncs in special, Python function overhead is avoided by vectorizing, that is, by passing an array to the function. Typically this approach works quite well, but sometimes it is more convenient to call a special function on scalar inputs inside a loop, for example when implementing your own ufunc. In this case the Python function overhead can become significant. Consider the following example:

```
import scipy.special as sc
cimport scipy.special.cython_special as csc

def python_tight_loop():
    cdef:
        int n
        double x = 1

    for n in range(100):
        sc.jv(n, x)

def cython_tight_loop():
    cdef:
        int n
        double x = 1

    for n in range(100):
        csc.jv(n, x)
```

On one computer `python_tight_loop` took about 131 microseconds to run and `cython_tight_loop` took about 18.2 microseconds to run. Obviously this example is contrived: one could just call `special.jv(np.`

arange(100), 1) and get results just as fast as in cython_tight_loop. The point is that if Python function overhead becomes significant in your code then the Cython bindings might be useful.

Releasing the GIL

One often needs to evaluate a special function at many points, and typically the evaluations are trivially parallelizable. Since the Cython bindings do not require the GIL, it is easy to run them in parallel using Cython's prange function. For example, suppose that we wanted to compute the fundamental solution to the Helmholtz equation:

$$\Delta_x G(x, y) + k^2 G(x, y) = \delta(x - y),$$

where k is the wavenumber and δ is the Dirac delta function. It is known that in two dimensions the unique (radiating) solution is

$$G(x, y) = \frac{i}{4} H_0^{(1)}(k|x - y|),$$

where $H_0^{(1)}$ is the Hankel function of the first kind, i.e. the function *hankel1*. The following example shows how we could compute this function in parallel:

```
from libc.math cimport fabs
cimport cython
from cython.parallel cimport prange

import numpy as np
import scipy.special as sc
cimport scipy.special.cython_special as csc

def serial_G(k, x, y):
    return 0.25j*sc.hankel1(0, k*np.abs(x - y))

@cython.boundscheck(False)
@cython.wraparound(False)
cdef void _parallel_G(double k, double[:,:] x, double[:,:] y,
                      double complex[:,:] out) nogil:
    cdef int i, j

    for i in prange(x.shape[0]):
        for j in range(y.shape[0]):
            out[i,j] = 0.25j*csc.hankel1(0, k*fabs(x[i,j] - y[i,j]))

def parallel_G(k, x, y):
    out = np.empty_like(x, dtype='complex128')
    _parallel_G(k, x, y, out)
    return out
```

(For help with compiling parallel code in Cython see here.) If the above Cython code is in a file test.pyx, then we can write an informal benchmark which compares the parallel and serial versions of the function:

```
import timeit

import numpy as np

from test import serial_G, parallel_G

def main():
    k = 1
    x, y = np.linspace(-100, 100, 1000), np.linspace(-100, 100, 1000)
    x, y = np.meshgrid(x, y)
```

```
    def serial():
        serial_G(k, x, y)

    def parallel():
        parallel_G(k, x, y)

    time_serial = timeit.timeit(serial, number=3)
    time_parallel = timeit.timeit(parallel, number=3)
    print("Serial method took {:.3} seconds".format(time_serial))
    print("Parallel method took {:.3} seconds".format(time_parallel))

if __name__ == "__main__":
    main()
```

On one quad-core computer the serial method took 1.29 seconds and the parallel method took 0.29 seconds.

Functions not in `scipy.special`

Some functions are not included in special because they are straightforward to implement with existing functions in NumPy and SciPy. To prevent reinventing the wheel, this section provides implementations of several such functions which hopefully illustrate how to handle similar functions. In all examples NumPy is imported as `np` and special is imported as `sc`.

The binary entropy function:

```
def binary_entropy(x):
    return -(sc.xlogy(x, x) + sc.xlog1py(1 - x, -x))/np.log(2)
```

The Heaviside step function:

```
def heaviside(x):
    return 0.5*(np.sign(x) + 1)
```

A similar idea can also be used to get a step function on [0, 1]:

```
def step(x):
    return 0.5*(np.sign(x) + np.sign(1 - x))
```

Translating and scaling can be used to get an arbitrary step function.

The ramp function:

```
def ramp(x):
    return np.maximum(0, x)
```

3.1.4 Integration (`scipy.integrate`)

The *`scipy.integrate`* sub-package provides several integration techniques including an ordinary differential equation integrator. An overview of the module is provided by the help command:

```
>>> help(integrate)
 Methods for Integrating Functions given function object.

   quad          -- General purpose integration.
```

```
   dblquad       -- General purpose double integration.
   tplquad       -- General purpose triple integration.
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature    -- Integrate with given tolerance using Gaussian quadrature.
   romberg       -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.

   trapz         -- Use trapezoidal rule to compute integral from samples.
   cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
   simps         -- Use Simpson's rule to compute integral from samples.
   romb          -- Use Romberg Integration to compute integral from
                    (2**k + 1) evenly-spaced samples.

   See the special module's orthogonal polynomials (special) for Gaussian
      quadrature roots and weights for other weighting factors and regions.

 Interface to numerical integrators of ODE systems.

   odeint        -- General integration of ordinary differential equations.
   ode           -- Integrate ODE using VODE and ZVODE routines.
```

General integration (quad)

The function *quad* is provided to integrate a function of one variable between two points. The points can be $\pm\infty$ ($\pm$ `inf`) to indicate infinite limits. For example, suppose you wish to integrate a bessel function `jv(2.5, x)` along the interval $[0, 4.5]$.

$$I = \int_0^{4.5} J_{2.5}\left(x\right)\, dx.$$

This could be computed using *quad*:

```
>>> import scipy.integrate as integrate
>>> import scipy.special as special
>>> result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
>>> result
(1.1178179380783249, 7.8663172481899801e-09)
```

```
>>> from numpy import sqrt, sin, cos, pi
>>> I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5) - 4.0/27*sqrt(2)*sin(4.5) +
...                 sqrt(2*pi) * special.fresnel(3/sqrt(pi))[0])
>>> I
1.117817938088701
```

```
>>> print(abs(result[0]-I))
1.03761443881e-11
```

The first argument to quad is a "callable" Python object (*i.e.* a function, method, or class instance). Notice the use of a lambda- function in this case as the argument. The next two arguments are the limits of integration. The return value is a tuple, with the first element holding the estimated value of the integral and the second element holding an upper bound on the error. Notice, that in this case, the true value of this integral is

$$I = \sqrt{\frac{2}{\pi}}\left(\frac{18}{27}\sqrt{2}\cos\left(4.5\right) - \frac{4}{27}\sqrt{2}\sin\left(4.5\right) + \sqrt{2\pi}\textrm{Si}\left(\frac{3}{\sqrt{\pi}}\right)\right),$$

where

$$\text{Si}(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right) dt.$$

is the Fresnel sine integral. Note that the numerically-computed integral is within 1.04×10^{-11} of the exact result — well below the reported error bound.

If the function to integrate takes additional parameters, the can be provided in the `args` argument. Suppose that the following integral shall be calculated:

$$I(a,b) = \int_0^1 ax^2 + b\,dx.$$

This integral can be evaluated by using the following code:

```
>>> from scipy.integrate import quad
>>> def integrand(x, a, b):
...     return a*x**2 + b
...
>>> a = 2
>>> b = 1
>>> I = quad(integrand, 0, 1, args=(a,b))
>>> I
(1.6666666666666667, 1.8503717077085944e-14)
```

Infinite inputs are also allowed in *quad* by using $\pm$ `inf` as one of the arguments. For example, suppose that a numerical value for the exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt.$$

is desired (and the fact that this integral can be computed as `special.expn(n,x)` is forgotten). The functionality of the function `special.expn` can be replicated by defining a new function `vec_expint` based on the routine *quad*:

```
>>> from scipy.integrate import quad
>>> def integrand(t, n, x):
...     return np.exp(-x*t) / t**n
...
```

```
>>> def expint(n, x):
...     return quad(integrand, 1, np.inf, args=(n, x))[0]
...
```

```
>>> vec_expint = np.vectorize(expint)
```

```
>>> vec_expint(3, np.arange(1.0, 4.0, 0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
>>> import scipy.special as special
>>> special.expn(3, np.arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
```

The function which is integrated can even use the quad argument (though the error bound may underestimate the error due to possible numerical error in the integrand from the use of *quad*). The integral in this case is

$$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n}\,dt\,dx = \frac{1}{n}.$$

```
>>> result = quad(lambda x: expint(3, x), 0, np.inf)
>>> print(result)
(0.33333333324560266, 2.8548934485373678e-09)
```

```
>>> I3 = 1.0/3.0
>>> print(I3)
0.333333333333
```

```
>>> print(I3 - result[0])
8.77306560731e-11
```

This last example shows that multiple integration can be handled using repeated calls to *quad*.

General multiple integration (dblquad, tplquad, nquad)

The mechanics for double and triple integration have been wrapped up into the functions *dblquad* and *tplquad*. These functions take the function to integrate and four, or six arguments, respectively. The limits of all inner integrals need to be defined as functions.

An example of using double integration to compute several values of I_n is shown below:

```
>>> from scipy.integrate import quad, dblquad
>>> def I(n):
...     return dblquad(lambda t, x: np.exp(-x*t)/t**n, 0, np.inf, lambda x: 1, lambda␣
↪x: np.inf)
...
```

```
>>> print(I(4))
(0.2500000000043577, 1.29830334693681e-08)
>>> print(I(3))
(0.33333333325010883, 1.3888461883425516e-08)
>>> print(I(2))
(0.4999999999985751, 1.3894083651858995e-08)
```

As example for non-constant limits consider the integral

$$I = \int_{y=0}^{1/2} \int_{x=0}^{1-2y} xy \, dx \, dy = \frac{1}{96}.$$

This integral can be evaluated using the expression below (Note the use of the non-constant lambda functions for the upper limit of the inner integral):

```
>>> from scipy.integrate import dblquad
>>> area = dblquad(lambda x, y: x*y, 0, 0.5, lambda x: 0, lambda x: 1-2*x)
>>> area
(0.010416666666666668, 1.1564823173178715e-16)
```

For n-fold integration, scipy provides the function *nquad*. The integration bounds are an iterable object: either a list of constant bounds, or a list of functions for the non-constant integration bounds. The order of integration (and therefore the bounds) is from the innermost integral to the outermost one.

The integral from above

$$I_n = \int_0^{\infty} \int_1^{\infty} \frac{e^{-xt}}{t^n} \, dt \, dx = \frac{1}{n}$$

can be calculated as

```
>>> from scipy import integrate
>>> N = 5
>>> def f(t, x):
...    return np.exp(-x*t) / t**N
...
>>> integrate.nquad(f, [[1, np.inf],[0, np.inf]])
(0.20000000000002294, 1.2239614263187945e-08)
```

Note that the order of arguments for *f* must match the order of the integration bounds; i.e. the inner integral with respect to t is on the interval $[1, \infty]$ and the outer integral with respect to x is on the interval $[0, \infty]$.

Non-constant integration bounds can be treated in a similar manner; the example from above

$$I = \int_{y=0}^{1/2} \int_{x=0}^{1-2y} xy \, dx \, dy = \frac{1}{96}.$$

can be evaluated by means of

```
>>> from scipy import integrate
>>> def f(x, y):
...     return x*y
...
>>> def bounds_y():
...     return [0, 0.5]
...
>>> def bounds_x(y):
...     return [0, 1-2*y]
...
>>> integrate.nquad(f, [bounds_x, bounds_y])
(0.010416666666666668, 4.101620128472366e-16)
```

which is the same result as before.

Gaussian quadrature

A few functions are also provided in order to perform simple Gaussian quadrature over a fixed interval. The first is *fixed_quad* which performs fixed-order Gaussian quadrature. The second function is *quadrature* which performs Gaussian quadrature of multiple orders until the difference in the integral estimate is beneath some tolerance supplied by the user. These functions both use the module `special.orthogonal` which can calculate the roots and quadrature weights of a large variety of orthogonal polynomials (the polynomials themselves are available as special functions returning instances of the polynomial class — e.g. *special.legendre*).

Romberg Integration

Romberg's method *[WPR]* is another method for numerically evaluating an integral. See the help function for *romberg* for further details.

Integrating using Samples

If the samples are equally-spaced and the number of samples available is $2^k + 1$ for some integer k, then Romberg *romb* integration can be used to obtain high-precision estimates of the integral using the available samples. Romberg integration uses the trapezoid rule at step-sizes related by a power of two and then performs Richardson extrapolation on these estimates to approximate the integral with a higher-degree of accuracy.

In case of arbitrary spaced samples, the two functions trapz (defined in numpy *[NPT]*) and `simps` are available. They are using Newton-Coates formulas of order 1 and 2 respectively to perform integration. The trapezoidal rule approximates the function as a straight line between adjacent points, while Simpson's rule approximates the function between three adjacent points as a parabola.

For an odd number of samples that are equally spaced Simpson's rule is exact if the function is a polynomial of order 3 or less. If the samples are not equally spaced, then the result is exact only if the function is a polynomial of order 2 or less.

```
>>> import numpy as np
>>> def f1(x):
...    return x**2
...
>>> def f2(x):
...    return x**3
...
>>> x = np.array([1,3,4])
>>> y1 = f1(x)
>>> from scipy.integrate import simps
>>> I1 = simps(y1, x)
>>> print(I1)
21.0
```

This corresponds exactly to

$$\int_1^4 x^2 \, dx = 21,$$

whereas integrating the second function

```
>>> y2 = f2(x)
>>> I2 = integrate.simps(y2, x)
>>> print(I2)
61.5
```

does not correspond to

$$\int_1^4 x^3 \, dx = 63.75$$

because the order of the polynomial in f2 is larger than two.

Faster integration using low-level callback functions

A user desiring reduced integration times may pass a C function pointer through `scipy.LowLevelCallable` to `quad`, `dblquad`, `tplquad` or `nquad` and it will be integrated and return a result in Python. The performance increase here arises from two factors. The primary improvement is faster function evaluation, which is provided by compilation of the function itself. Additionally we have a speedup provided by the removal of function calls between C and Python in `quad`. This method may provide a speed improvements of ~2x for trivial functions such as sine but can produce a much more noticeable improvements (10x+) for more complex functions. This feature then, is geared towards a user with numerically intensive integrations willing to write a little C to reduce computation time significantly.

The approach can be used, for example, via `ctypes` in a few simple steps:

1.) Write an integrand function in C with the function signature `double f(int n, double *x, void *user_data)`, where `x` is an array containing the point the function f is evaluated at, and `user_data` to arbitrary additional data you want to provide.

```
/* testlib.c */
double f(int n, double *x, void *user_data) {
    double c = *(double *)user_data;
    return c + x[0] - x[1] * x[2]; /* corresponds to c + x - y * z */
}
```

2.) Now compile this file to a shared/dynamic library (a quick search will help with this as it is OS-dependent). The user must link any math libraries, etc. used. On linux this looks like:

```
$ gcc -shared -fPIC -o testlib.so testlib.c
```

The output library will be referred to as `testlib.so`, but it may have a different file extension. A library has now been created that can be loaded into Python with `ctypes`.

3.) Load shared library into Python using `ctypes` and set `restypes` and `argtypes` - this allows Scipy to interpret the function correctly:

```
import os, ctypes
from scipy import integrate, LowLevelCallable

lib = ctypes.CDLL(os.path.abspath('testlib.so'))
lib.f.restype = ctypes.c_double
lib.f.argtypes = (ctypes.c_int, ctypes.POINTER(ctypes.c_double), ctypes.c_void_p)

c = ctypes.c_double(1.0)
user_data = ctypes.cast(ctypes.pointer(c), ctypes.c_void_p)

func = LowLevelCallable(lib.f, user_data)
```

The last `void *user_data` in the function is optional and can be omitted (both in the C function and ctypes argtypes) if not needed. Note that the coordinates are passed in as an array of doubles rather than a separate argument.

4.) Now integrate the library function as normally, here using *nquad*:

```
>>> integrate.nquad(func, [[0, 10], [-10, 0], [-1, 1]])
(1200.0, 1.1102230246251565e-11)
```

The Python tuple is returned as expected in a reduced amount of time. All optional parameters can be used with this method including specifying singularities, infinite bounds, etc.

Ordinary differential equations (`odeint`)

Integrating a set of ordinary differential equations (ODEs) given initial conditions is another useful example. The function *odeint* is available in SciPy for integrating a first-order vector differential equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right),$$

given initial conditions $\mathbf{y}\left(0\right) = y_0$, where $\mathbf{y}$ is a length N vector and $\mathbf{f}$ is a mapping from $\mathcal{R}^N$ to $\mathcal{R}^N$. A higher-order ordinary differential equation can always be reduced to a differential equation of this type by introducing intermediate derivatives into the $\mathbf{y}$ vector.

For example suppose it is desired to find the solution to the following second-order differential equation:

$$\frac{d^2 w}{dz^2} - zw(z) = 0$$

with initial conditions $w(0) = \frac{1}{\sqrt[3]{3^2}\Gamma\left(\frac{2}{3}\right)}$ and $\left.\frac{dw}{dz}\right|_{z=0} = -\frac{1}{\sqrt[3]{3}\Gamma\left(\frac{1}{3}\right)}$. It is known that the solution to this differential equation with these boundary conditions is the Airy function

$$w = \mathrm{Ai}(z),$$

which gives a means to check the integrator using `special.airy`.

First, convert this ODE into standard form by setting $\mathbf{y} = \left[\frac{dw}{dz}, w\right]$ and $t = z$. Thus, the differential equation becomes

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} ty_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \mathbf{y}.$$

In other words,

$$\mathbf{f}(\mathbf{y}, t) = \mathbf{A}(t)\,\mathbf{y}.$$

As an interesting reminder, if $\mathbf{A}(t)$ commutes with $\int_0^t \mathbf{A}(\tau)\, d\tau$ under matrix multiplication, then this linear differential equation has an exact solution using the matrix exponential:

$$\mathbf{y}(t) = \exp\left(\int_0^t \mathbf{A}(\tau)\, d\tau\right) \mathbf{y}(0),$$

However, in this case, $\mathbf{A}(t)$ and its integral do not commute.

There are many optional inputs and outputs available when using odeint which can help tune the solver. These additional inputs and outputs are not needed much of the time, however, and the three required input arguments and the output solution suffice. The required inputs are the function defining the derivative, *fprime*, the initial conditions vector, *y0*, and the time points to obtain a solution, *t*, (with the initial value point as the first element of this sequence). The output to `odeint` is a matrix where each row contains the solution vector at each requested time point (thus, the initial conditions are given in the first output row).

The following example illustrates the use of odeint including the usage of the *Dfun* option which allows the user to specify a gradient (with respect to $\mathbf{y}$) of the function, $\mathbf{f}(\mathbf{y}, t)$.

```
>>> from scipy.integrate import odeint
>>> from scipy.special import gamma, airy
>>> y1_0 = 1.0 / 3**(2.0/3.0) / gamma(2.0/3.0)
>>> y0_0 = -1.0 / 3**(1.0/3.0) / gamma(1.0/3.0)
>>> y0 = [y0_0, y1_0]
>>> def func(y, t):
...     return [t*y[1],y[0]]
...
```

```
>>> def gradient(y, t):
...     return [[0,t], [1,0]]
...
```

```
>>> x = np.arange(0, 4.0, 0.01)
>>> t = x
>>> ychk = airy(x)[0]
>>> y = odeint(func, y0, t)
>>> y2 = odeint(func, y0, t, Dfun=gradient)
```

```
>>> ychk[:36:6]
array([0.355028, 0.339511, 0.324068, 0.308763, 0.293658, 0.278806])
```

```
>>> y[:36:6,1]
array([0.355028, 0.339511, 0.324067, 0.308763, 0.293658, 0.278806])
```

```
>>> y2[:36:6,1]
array([0.355028, 0.339511, 0.324067, 0.308763, 0.293658, 0.278806])
```

Solving a system with a banded Jacobian matrix

`odeint` can be told that the Jacobian is *banded*. For a large system of differential equations that are known to be stiff, this can improve performance significantly.

As an example, we'll solve the one-dimensional Gray-Scott partial differential equations using the method of lines *[MOL]*. The Gray-Scott equations for the functions $u(x,t)$ and $v(x,t)$ on the interval $x \in [0, L]$ are

$$\begin{aligned}\frac{\partial u}{\partial t} &= D_u \frac{\partial^2 u}{\partial x^2} - uv^2 + f(1-u) \\ \frac{\partial v}{\partial t} &= D_v \frac{\partial^2 v}{\partial x^2} + uv^2 - (f+k)v\end{aligned}$$

where D_u and D_v are the diffusion coefficients of the components u and v, respectively, and f and k are constants. (For more information about the system, see http://groups.csail.mit.edu/mac/projects/amorphous/GrayScott/)

We'll assume Neumann (i.e. "no flux") boundary conditions:

$$\frac{\partial u}{\partial x}(0,t) = 0, \quad \frac{\partial v}{\partial x}(0,t) = 0, \quad \frac{\partial u}{\partial x}(L,t) = 0, \quad \frac{\partial v}{\partial x}(L,t) = 0$$

To apply the method of lines, we discretize the x variable by defining the uniformly spaced grid of N points $\{x_0, x_1, \ldots, x_{N-1}\}$, with $x_0 = 0$ and $x_{N-1} = L$. We define $u_j(t) \equiv u(x_k, t)$ and $v_j(t) \equiv v(x_k, t)$, and replace the x derivatives with finite differences. That is,

$$\frac{\partial^2 u}{\partial x^2}(x_j, t) \rightarrow \frac{u_{j-1}(t) - 2u_j(t) + u_{j+1}(t)}{(\Delta x)^2}$$

We then have a system of $2N$ ordinary differential equations:

$$\begin{aligned}\frac{du_j}{dt} &= \frac{D_u}{(\Delta x)^2}\left(u_{j-1} - 2u_j + u_{j+1}\right) - u_j v_j^2 + f(1-u_j) \\ \frac{dv_j}{dt} &= \frac{D_v}{(\Delta x)^2}\left(v_{j-1} - 2v_j + v_{j+1}\right) + u_j v_j^2 - (f+k)v_j\end{aligned} \tag{3.1}$$

For convenience, the (t) arguments have been dropped.

To enforce the boundary conditions, we introduce "ghost" points x_{-1} and x_N, and define $u_{-1}(t) \equiv u_1(t)$, $u_N(t) \equiv u_{N-2}(t)$; $v_{-1}(t)$ and $v_N(t)$ are defined analogously.

Then

$$\begin{aligned}\frac{du_0}{dt} &= \frac{D_u}{(\Delta x)^2}\left(2u_1 - 2u_0\right) - u_0 v_0^2 + f(1-u_0) \\ \frac{dv_0}{dt} &= \frac{D_v}{(\Delta x)^2}\left(2v_1 - 2v_0\right) + u_0 v_0^2 - (f+k)v_0\end{aligned} \tag{3.2}$$

and

$$\begin{aligned}\frac{du_{N-1}}{dt} &= \frac{D_u}{(\Delta x)^2}\left(2u_{N-2} - 2u_{N-1}\right) - u_{N-1} v_{N-1}^2 + f(1-u_{N-1}) \\ \frac{dv_{N-1}}{dt} &= \frac{D_v}{(\Delta x)^2}\left(2v_{N-2} - 2v_{N-1}\right) + u_{N-1} v_{N-1}^2 - (f+k)v_{N-1}\end{aligned} \tag{3.3}$$

Our complete system of $2N$ ordinary differential equations is (**??**) for $k = 1, 2, \ldots, N-2$, along with (**??**) and (**??**).

We can now starting implementing this system in code. We must combine $\{u_k\}$ and $\{v_k\}$ into a single vector of length $2N$. The two obvious choices are $\{u_0, u_1, \ldots, u_{N-1}, v_0, v_1, \ldots, v_{N-1}\}$ and $\{u_0, v_0, u_1, v_1, \ldots, u_{N-1}, v_{N-1}\}$. Mathematically, it does not matter, but the choice affects how efficiently *odeint* can solve the system. The reason is in how the order affects the pattern of the nonzero elements of the Jacobian matrix.

When the variables are ordered as $\{u_0, u_1, \ldots, u_{N-1}, v_0, v_1, \ldots, v_{N-1}\}$, the pattern of nonzero elements of the Jacobian matrix is

```
* * 0 0 0 0 0 * 0 0 0 0 0 0
* * * 0 0 0 0 0 * 0 0 0 0 0
0 * * * 0 0 0 0 0 * 0 0 0 0
0 0 * * * 0 0 0 0 0 * 0 0 0
0 0 0 * * * 0 0 0 0 0 * 0 0
0 0 0 0 * * * 0 0 0 0 0 * 0
0 0 0 0 0 * * 0 0 0 0 0 0 *
* 0 0 0 0 0 0 * * 0 0 0 0 0
0 * 0 0 0 0 0 * * * 0 0 0 0
0 0 * 0 0 0 0 0 * * * 0 0 0
0 0 0 * 0 0 0 0 0 * * * 0 0
0 0 0 0 * 0 0 0 0 0 * * * 0
0 0 0 0 0 * 0 0 0 0 0 * * *
0 0 0 0 0 0 * 0 0 0 0 ) * *
```

The Jacobian pattern with variables interleaved as $\{u_0, v_0, u_1, v_1, \ldots, u_{N-1}, v_{N-1}\}$ is

```
* * * 0 0 0 0 0 0 0 0 0 0 0
* * 0 * 0 0 0 0 0 0 0 0 0 0
* 0 * * * 0 0 0 0 0 0 0 0 0
0 * * * 0 * 0 0 0 0 0 0 0 0
0 0 * 0 * * * 0 0 0 0 0 0 0
0 0 0 * * * 0 * 0 0 0 0 0 0
0 0 0 0 * 0 * * * 0 0 0 0 0
0 0 0 0 0 * * * 0 * 0 0 0 0
0 0 0 0 0 0 * 0 * * * 0 0 0
0 0 0 0 0 0 0 * * * 0 * 0 0
0 0 0 0 0 0 0 0 * 0 * * * 0
0 0 0 0 0 0 0 0 0 * * * 0 *
0 0 0 0 0 0 0 0 0 0 * 0 * *
0 0 0 0 0 0 0 0 0 0 0 * * *
```

In both cases, there are just five nontrivial diagonals, but when the variables are interleaved, the bandwidth is much smaller. That is, the main diagonal and the two diagonals immediately above and the two immediately below the main diagonal are the nonzero diagonals. This is important, because the inputs `mu` and `ml` of *odeint* are the upper and lower bandwidths of the Jacobian matrix. When the variables are interleaved, `mu` and `ml` are 2. When the variables are stacked with $\{v_k\}$ following $\{u_k\}$, the upper and lower bandwidths are N.

With that decision made, we can write the function that implements the system of differential equations.

First, we define the functions for the source and reaction terms of the system:

```python
def G(u, v, f, k):
    return f * (1 - u) - u*v**2

def H(u, v, f, k):
    return -(f + k) * v + u*v**2
```

Next we define the function that computes the right-hand-side of the system of differential equations:

```python
def grayscott1d(y, t, f, k, Du, Dv, dx):
    """
    Differential equations for the 1D Gray-Scott equations.

    The ODEs are derived using the method of lines.
    """
    # The vectors u and v are interleaved in y.  We define
    # views of u and v by slicing y.
    u = y[::2]
    v = y[1::2]

    # dydt is the return value of this function.
    dydt = np.empty_like(y)
```

```
    # Just like u and v are views of the interleaved vectors
    # in y, dudt and dvdt are views of the interleaved output
    # vectors in dydt.
    dudt = dydt[::2]
    dvdt = dydt[1::2]

    # Compute du/dt and dv/dt.  The end points and the interior points
    # are handled separately.
    dudt[0]    = G(u[0],    v[0],    f, k) + Du * (-2.0*u[0] + 2.0*u[1]) / dx**2
    dudt[1:-1] = G(u[1:-1], v[1:-1], f, k) + Du * np.diff(u,2) / dx**2
    dudt[-1]   = G(u[-1],   v[-1],   f, k) + Du * (- 2.0*u[-1] + 2.0*u[-2]) / dx**2
    dvdt[0]    = H(u[0],    v[0],    f, k) + Dv * (-2.0*v[0] + 2.0*v[1]) / dx**2
    dvdt[1:-1] = H(u[1:-1], v[1:-1], f, k) + Dv * np.diff(v,2) / dx**2
    dvdt[-1]   = H(u[-1],   v[-1],   f, k) + Dv * (-2.0*v[-1] + 2.0*v[-2]) / dx**2

    return dydt
```

We won't implement a function to compute the Jacobian, but we will tell *odeint* that the Jacobian matrix is banded. This allows the underlying solver (LSODA) to avoid computing values that it knows are zero. For a large system, this improves the performance significantly, as demonstrated in the following ipython session.

First, we define the required inputs:

```
In [31]: y0 = np.random.randn(5000)

In [32]: t = np.linspace(0, 50, 11)

In [33]: f = 0.024

In [34]: k = 0.055

In [35]: Du = 0.01

In [36]: Dv = 0.005

In [37]: dx = 0.025
```

Time the computation without taking advantage of the banded structure of the Jacobian matrix:

```
In [38]: %timeit sola = odeint(grayscott1d, y0, t, args=(f, k, Du, Dv, dx))
1 loop, best of 3: 25.2 s per loop
```

Now set `ml=2` and `mu=2`, so *odeint* knows that the Jacobian matrix is banded:

```
In [39]: %timeit solb = odeint(grayscott1d, y0, t, args=(f, k, Du, Dv, dx), ml=2,␣
↪mu=2)
10 loops, best of 3: 191 ms per loop
```

That is quite a bit faster!

Let's ensure that they have computed the same result:

```
In [41]: np.allclose(sola, solb)
Out[41]: True
```

References

3.1.5 Optimization (`scipy.optimize`)

The *scipy.optimize* package provides several commonly used optimization algorithms. A detailed listing is available: *scipy.optimize* (can also be found by `help(scipy.optimize)`).

The module contains:

1. Unconstrained and constrained minimization of multivariate scalar functions (*minimize*) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP)
2. Global (brute-force) optimization routines (e.g. *basinhopping*, *differential_evolution*)
3. Least-squares minimization (*least_squares*) and curve fitting (*curve_fit*) algorithms
4. Scalar univariate functions minimizers (*minimize_scalar*) and root finders (*newton*)
5. Multivariate equation system solvers (*root*) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov).

Below, several examples demonstrate their basic usage.

Unconstrained minimization of multivariate scalar functions (`minimize`)

The *minimize* function provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in *scipy.optimize*. To demonstrate the minimization function consider the problem of minimizing the Rosenbrock function of N variables: $f(\mathbf{x}) = \sum_{i=1}^{N-1} 100\left(x_i - x_{i-1}^2\right)^2 + \left(1 - x_{i-1}\right)^2 .The minimum value of this function is 0 which is achieved when x_i = 1.$

Note that the Rosenbrock function and its derivatives are included in *scipy.optimize*. The implementations shown in the following sections provide examples of how to define an objective function as well as its jacobian and hessian functions.

Nelder-Mead Simplex algorithm (`method='Nelder-Mead'`)

In the example below, the *minimize* routine is used with the *Nelder-Mead* simplex algorithm (selected through the `method` parameter):

```
>>> import numpy as np
>>> from scipy.optimize import minimize
```

```
>>> def rosen(x):
...     """The Rosenbrock function"""
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
```

```
>>> x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
>>> res = minimize(rosen, x0, method='nelder-mead',
...                options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
```

```
>>> print(res.x)
[ 1.  1.  1.  1.  1.]
```

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

Another optimization algorithm that needs only function calls to find the minimum is *Powell*'s method available by setting `method='powell'` in *minimize*.

Broyden-Fletcher-Goldfarb-Shanno algorithm (`method='BFGS'`)

In order to converge more quickly to the solution, this routine uses the gradient of the objective function. If the gradient is not given by the user, then it is estimated using first-differences. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method typically requires fewer function calls than the simplex algorithm even when the gradient must be estimated.

To demonstrate this algorithm, the Rosenbrock function is again used. The gradient of the Rosenbrock function is the vector:

$$\begin{aligned} \frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1 - x_{i-1}\right)\delta_{i-1,j}. \\ &= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1 - x_j\right). \end{aligned}$$

This expression is valid for the interior derivatives. Special cases are

$$\begin{aligned} \frac{\partial f}{\partial x_0} &= -400x_0\left(x_1 - x_0^2\right) - 2\left(1 - x_0\right), \\ \frac{\partial f}{\partial x_{N-1}} &= 200\left(x_{N-1} - x_{N-2}^2\right). \end{aligned}$$

A Python function which computes this gradient is constructed by the code-segment:

```
>>> def rosen_der(x):
...     xm = x[1:-1]
...     xm_m1 = x[:-2]
...     xm_p1 = x[2:]
...     der = np.zeros_like(x)
...     der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
...     der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
...     der[-1] = 200*(x[-1]-x[-2]**2)
...     return der
```

This gradient information is specified in the *minimize* function through the `jac` parameter as illustrated below.

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 51                     # may vary
         Function evaluations: 63
         Gradient evaluations: 63
>>> res.x
array([1., 1., 1., 1., 1.])
```

Newton-Conjugate-Gradient algorithm (`method='Newton-CG'`)

The method which requires the fewest function calls and is therefore often the fastest method to minimize functions of many variables uses the Newton-Conjugate Gradient algorithm. This method is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Newton's method is based on fitting the function locally to a quadratic form: $\mathrm{f}(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) .where \mathbf{H}(\mathbf{x}_0)$ is a matrix of second-derivatives (the Hessian). If the Hessian is

positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero, resulting in $\mathrm{x}_{\mathrm{opt}} = \mathbf{x}_0 - \mathbf{H}^{-1}\nabla f. TheinverseoftheHessianisevaluatedusingtheconjugate - gradientmethod. AnexampleofemployingthismethodtominimizingtheRosenbrockfunctionisgivenbelow. Totakefulladvanta$ $CGmethod, afunctionwhichcomputestheHessianmustbeprovided. TheHessianmatrixitselfdoesnotneedtobeconstructed, on$

Full Hessian example:

The Hessian of the Rosenbrock function is

$$
\begin{aligned}
H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} &= 200\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 400x_i\left(\delta_{i+1,j} - 2x_i\delta_{i,j}\right) - 400\delta_{i,j}\left(x_{i+1} - x_i^2\right) + 2\delta_{i,j}, \\
&= \left(202 + 1200x_i^2 - 400x_{i+1}\right)\delta_{i,j} - 400x_i\delta_{i+1,j} - 400x_{i-1}\delta_{i-1,j},
\end{aligned}
$$

if $i, j \in [1, N-2]$ with $i, j \in [0, N-1]$ defining the $N \times N$ matrix. Other non-zero entries of the matrix are

$$
\begin{aligned}
\frac{\partial^2 f}{\partial x_0^2} &= 1200x_0^2 - 400x_1 + 2, \\
\frac{\partial^2 f}{\partial x_0 \partial x_1} = \frac{\partial^2 f}{\partial x_1 \partial x_0} &= -400x_0, \\
\frac{\partial^2 f}{\partial x_{N-1} \partial x_{N-2}} = \frac{\partial^2 f}{\partial x_{N-2} \partial x_{N-1}} &= -400x_{N-2}, \\
\frac{\partial^2 f}{\partial x_{N-1}^2} &= 200.
\end{aligned}
$$

For example, the Hessian when $N = 5$ is H= $\left[\begin{array}{cccc} 1200x_0^2 - 400x_1 + 2 & -400x_0 & 0 & 0 \\ -400x_0 & 202 + 1200x_1^2 - 400x_2 & -400x_1 & 0 \\ 0 & -400x_1 & 202 + 1200x_2^2 - 400x_3 & -40 \\ 0 & & -400x_2 & 202 + 1200x \\ 0 & 0 & 0 & -40 \end{array}\right.$

$CGmethodisshowninthefollowingexample:$

```
>>> def rosen_hess(x):
...     x = np.asarray(x)
...     H = np.diag(-400*x[:-1],1) - np.diag(400*x[:-1],-1)
...     diagonal = np.zeros_like(x)
...     diagonal[0] = 1200*x[0]**2-400*x[1]+2
...     diagonal[-1] = 200
...     diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
...     H = H + np.diag(diagonal)
...     return H
```

```
>>> res = minimize(rosen, x0, method='Newton-CG',
...                jac=rosen_der, hess=rosen_hess,
...                options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 19                       # may vary
         Function evaluations: 22
         Gradient evaluations: 19
         Hessian evaluations: 19
>>> res.x
array([1.,  1.,  1.,  1.,  1.])
```

Hessian product example:

For larger minimization problems, storing the entire Hessian matrix can consume considerable time and memory. The Newton-CG algorithm only needs the product of the Hessian times an arbitrary vector. As a result, the user can supply code to compute this product rather than the full Hessian by giving a `hess` function which take the minimization vector as the first argument and the arbitrary vector as the second argument (along with extra arguments passed to the function to be minimized). If possible, using Newton-CG with the Hessian product option is probably the fastest way to minimize the function.

In this case, the product of the Rosenbrock Hessian with an arbitrary vector is not difficult to compute. If $\mathbf{p}$ is the arbitrary vector, then $\mathbf{H}(\mathbf{x})\,\mathbf{p}$ has elements: $\mathrm{H}(\mathbf{x})\,\mathbf{p} =$

$$\left[\begin{array}{c} \left(1200x_{0}^{2}-400x_{1}+2\right)p_{0}-400x_{0}p_{1}\\ \vdots\\ -400x_{i-1}p_{i-1}+\left(202+1200x_{i}^{2}-400x_{i+1}\right)p_{i}-400x_{i}p_{i+1}\\ \vdots\\ -400x_{N-2}p_{N-2}+200p_{N-1}\end{array}\right] .CodewhichmakesuseofthisHessianproducttominimizetheF$$

```
>>> def rosen_hess_p(x, p):
...     x = np.asarray(x)
...     Hp = np.zeros_like(x)
...     Hp[0] = (1200*x[0]**2 - 400*x[1] + 2)*p[0] - 400*x[0]*p[1]
...     Hp[1:-1] = -400*x[:-2]*p[:-2]+(202+1200*x[1:-1]**2-400*x[2:])*p[1:-1] \
...                -400*x[1:-1]*p[2:]
...     Hp[-1] = -400*x[-2]*p[-2] + 200*p[-1]
...     return Hp
```

```
>>> res = minimize(rosen, x0, method='Newton-CG',
...                jac=rosen_der, hessp=rosen_hess_p,
...                options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20                    # may vary
         Function evaluations: 23
         Gradient evaluations: 20
         Hessian evaluations: 44
>>> res.x
array([1., 1., 1., 1., 1.])
```

Constrained minimization of multivariate scalar functions (`minimize`)

The *minimize* function also provides an interface to several constrained minimization algorithm. As an example, the Sequential Least SQuares Programming optimization algorithm (SLSQP) will be considered here. This algorithm allows to deal with constrained minimization problems of the form:

$$\begin{array}{rll} \min F(x) & & \\ \text{subject to} & C_j(X) = 0, & j = 1, ..., \text{MEQ} \\ & C_j(x) \geq 0, & j = \text{MEQ}+1, ..., M \\ & XL \leq x \leq XU, & I = 1, ..., N. \end{array}$$

As an example, let us consider the problem of maximizing the function: f(x, y) = 2 x y + 2 x - x$^2 - 2y^2 subject to an equality and an inequality constraints defined as: to$ $x^3 - y = 0$ $y - 1 \geq 0$The objective function and its derivative are defined as follows.

```
>>> def func(x, sign=1.0):
...     """ Objective function """
...     return sign*(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)
```

```
>>> def func_deriv(x, sign=1.0):
...     """ Derivative of objective function """
...     dfdx0 = sign*(-2*x[0] + 2*x[1] + 2)
...     dfdx1 = sign*(2*x[0] - 4*x[1])
...     return np.array([ dfdx0, dfdx1 ])
```

Note that since `minimize` only minimizes functions, the `sign` parameter is introduced to multiply the objective function (and its derivative) by -1 in order to perform a maximization.

Then constraints are defined as a sequence of dictionaries, with keys `type`, `fun` and `jac`.

```
>>> cons = ({'type': 'eq',
...          'fun' : lambda x: np.array([x[0]**3 - x[1]]),
...          'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
...         {'type': 'ineq',
...          'fun' : lambda x: np.array([x[1] - 1]),
...          'jac' : lambda x: np.array([0.0, 1.0])})
```

Now an unconstrained optimization can be performed as:

```
>>> res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
...                method='SLSQP', options={'disp': True})
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -2.0
            Iterations: 4                              # may vary
            Function evaluations: 5
            Gradient evaluations: 4
>>> print(res.x)
[ 2.  1.]
```

and a constrained optimization as:

```
>>> res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
...                constraints=cons, method='SLSQP', options={'disp': True})
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -1.00000018311
            Iterations: 9                                  # may vary
            Function evaluations: 14
            Gradient evaluations: 9
>>> print(res.x)
[ 1.00000009  1.        ]
```

Least-squares minimization (`least_squares`)

SciPy is capable of solving robustified bound constrained nonlinear least-squares problems:

$$\min_{\mathbf{x}} \frac{1}{2} \sum_{i=1}^{m} \rho\left(f_i(\mathbf{x})^2\right) \tag{3.4}$$

$$\text{subject to } \mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \tag{3.5}$$

Here $f_i(\mathbf{x})$ are smooth functions from $\mathbb{R}^n$ to $\mathbb{R}$, we refer to them as residuals. The purpose of a scalar valued function $\rho(\cdot)$ is to reduce the influence of outlier residuals and contribute to robustness of the solution, we refer to it as a loss

function. A linear loss function gives a standard least-squares problem. Additionally, constraints in a form of lower and upper bounds on some of x_j are allowed.

All methods specific to least-squares minimization utilize a $m \times n$ matrix of partial derivatives called Jacobian and defined as $J_{ij} = \partial f_i / \partial x_j$. It is highly recommended to compute this matrix analytically and pass it to `least_squares`, otherwise it will be estimated by finite differences which takes a lot of additional time and can be very inaccurate in hard cases.

Function `least_squares` can be used for fitting a function $\varphi(t; \mathbf{x})$ to empirical data $\{(t_i, y_i), i = 0, \ldots, m-1\}$. To do this one should simply precompute residuals as $f_i(\mathbf{x}) = w_i(\varphi(t_i; \mathbf{x}) - y_i)$, where w_i are weights assigned to each observation.

Example of solving a fitting problem

Here we consider "Analysis of an Enzyme Reaction" problem formulated in[1]. There are 11 residuals defined as

$$f_i(x) = \frac{x_0(u_i^2 + u_i x_1)}{u_i^2 + u_i x_2 + x3} - y_i, \quad i = 0, \ldots, 10,$$

where y_i are measurement values and u_i are values of the independent variable. The unknown vector of parameters is $\mathbf{x} = (x_0, x_1, x_2, x_3)^T$. As was said previously, it is recommended to compute Jacobian matrix in a closed form:

$$J_{i0} = \frac{\partial f_i}{\partial x_0} = \frac{u_i^2 + u_i x_1}{u_i^2 + u_i x_2 + x_3} \quad (3.6)$$

$$J_{i1} = \frac{\partial f_i}{\partial x_1} = \frac{u_i x_0}{u_i^2 + u_i x_2 + x_3} \quad (3.7)$$

$$J_{i2} = \frac{\partial f_i}{\partial x_2} = -\frac{x_0(u_i^2 + u_i x_1) u_i}{(u_i^2 + u_i x_2 + x_3)^2} \quad (3.8)$$

$$J_{i3} = \frac{\partial f_i}{\partial x_3} = -\frac{x_0(u_i^2 + u_i x_1)}{(u_i^2 + u_i x_2 + x_3)^2} \quad (3.9)$$

We are going to use the "hard" starting point defined in[1]. To find a physically meaningful solution, avoid potential division by zero and assure convergence to the global minimum we impose constraints $0 \leq x_j \leq 100, j = 0, 1, 2, 3$.

The code below implements least-squares estimation of $\mathbf{x}$ and finally plots the original data and the fitted model function:

```
>>> from scipy.optimize import least_squares
```

```
>>> def model(x, u):
...     return x[0] * (u ** 2 + x[1] * u) / (u ** 2 + x[2] * u + x[3])
```

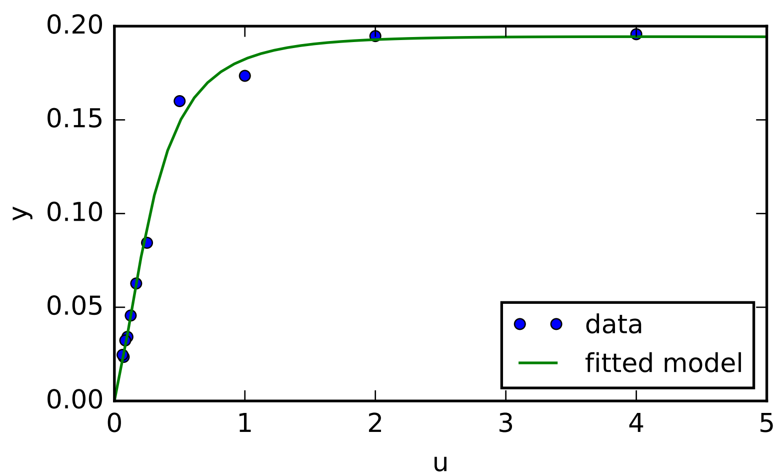
```
>>> def fun(x, u, y):
...     return model(x, u) - y
```

```
>>> def jac(x, u, y):
...     J = np.empty((u.size, x.size))
...     den = u ** 2 + x[2] * u + x[3]
...     num = u ** 2 + x[1] * u
...     J[:, 0] = num / den
...     J[:, 1] = x[0] * u / den
...     J[:, 2] = -x[0] * num * u / den ** 2
...     J[:, 3] = -x[0] * num / den ** 2
...     return J
```

[1] Brett M. Averick et al., "The MINPACK-2 Test Problem Collection".

```
>>> u = np.array([4.0, 2.0, 1.0, 5.0e-1, 2.5e-1, 1.67e-1, 1.25e-1, 1.0e-1,
...               8.33e-2, 7.14e-2, 6.25e-2])
>>> y = np.array([1.957e-1, 1.947e-1, 1.735e-1, 1.6e-1, 8.44e-2, 6.27e-2,
...               4.56e-2, 3.42e-2, 3.23e-2, 2.35e-2, 2.46e-2])
>>> x0 = np.array([2.5, 3.9, 4.15, 3.9])
>>> res = least_squares(fun, x0, jac=jac, bounds=(0, 100), args=(u, y), verbose=1)
`ftol` termination condition is satisfied.
Function evaluations 130, initial cost 4.4383e+00, final cost 1.5375e-04, first-order
↪optimality 4.92e-08.
>>> res.x
array([ 0.19280596,  0.19130423,  0.12306063,  0.13607247])
```

```
>>> import matplotlib.pyplot as plt
>>> u_test = np.linspace(0, 5)
>>> y_test = model(res.x, u_test)
>>> plt.plot(u, y, 'o', markersize=4, label='data')
>>> plt.plot(u_test, y_test, label='fitted model')
>>> plt.xlabel("u")
>>> plt.ylabel("y")
>>> plt.legend(loc='lower right')
>>> plt.show()
```



Further examples

Three interactive examples below illustrate usage of *least_squares* in greater detail.

1. Large-scale bundle adjustment in scipy demonstrates large-scale capabilities of *least_squares* and how to efficiently compute finite difference approximation of sparse Jacobian.
2. Robust nonlinear regression in scipy shows how to handle outliers with a robust loss function in a nonlinear regression.
3. Solving a discrete boundary-value problem in scipy examines how to solve a large system of equations and use bounds to achieve desired properties of the solution.

For the details about mathematical algorithms behind the implementation refer to documentation of *least_squares*.

Univariate function minimizers (`minimize_scalar`)

Often only the minimum of an univariate function (i.e. a function that takes a scalar as input) is needed. In these circumstances, other optimization techniques have been developed that can work faster. These are accessible from the `minimize_scalar` function which proposes several algorithms.

Unconstrained minimization (`method='brent'`)

There are actually two methods that can be used to minimize an univariate function: `brent` and `golden`, but `golden` is included only for academic purposes and should rarely be used. These can be respectively selected through the *method* parameter in `minimize_scalar`. The `brent` method uses Brent's algorithm for locating a minimum. Optimally a bracket (the `bracket` parameter) should be given which contains the minimum desired. A bracket is a triple (a, b, c) such that $f(a) > f(b) < f(c)$ and $a < b < c$. If this is not given, then alternatively two starting points can be chosen and a bracket will be found from these points using a simple marching algorithm. If these two starting points are not provided *0* and *1* will be used (this may not be the right choice for your function and result in an unexpected minimum being returned).

Here is an example:

```
>>> from scipy.optimize import minimize_scalar
>>> f = lambda x: (x - 2) * (x + 1)**2
>>> res = minimize_scalar(f, method='brent')
>>> print(res.x)
1.0
```

Bounded minimization (`method='bounded'`)

Very often, there are constraints that can be placed on the solution space before minimization occurs. The *bounded* method in `minimize_scalar` is an example of a constrained minimization procedure that provides a rudimentary interval constraint for scalar functions. The interval constraint allows the minimization to occur only between two fixed endpoints, specified using the mandatory *bounds* parameter.

For example, to find the minimum of $J_1(x)$ near $x = 5$, `minimize_scalar` can be called using the interval $[4, 7]$ as a constraint. The result is $x_{\text{min}} = 5.3314$:

```
>>> from scipy.special import j1
>>> res = minimize_scalar(j1, bounds=(4, 7), method='bounded')
>>> res.x
5.33144184241
```

Custom minimizers

Sometimes, it may be useful to use a custom method as a (multivariate or univariate) minimizer, for example when using some library wrappers of `minimize` (e.g. `basinhopping`).

We can achieve that by, instead of passing a method name, we pass a callable (either a function or an object implementing a *__call__* method) as the *method* parameter.

Let us consider an (admittedly rather virtual) need to use a trivial custom multivariate minimization method that will just search the neighborhood in each dimension independently with a fixed step size:

```
>>> from scipy.optimize import OptimizeResult
>>> def custmin(fun, x0, args=(), maxfev=None, stepsize=0.1,
...         maxiter=100, callback=None, **options):
...     bestx = x0
...     besty = fun(x0)
...     funcalls = 1
```

```
...     niter = 0
...     improved = True
...     stop = False
...
...     while improved and not stop and niter < maxiter:
...         improved = False
...         niter += 1
...         for dim in range(np.size(x0)):
...             for s in [bestx[dim] - stepsize, bestx[dim] + stepsize]:
...                 testx = np.copy(bestx)
...                 testx[dim] = s
...                 testy = fun(testx, *args)
...                 funcalls += 1
...                 if testy < besty:
...                     besty = testy
...                     bestx = testx
...                     improved = True
...             if callback is not None:
...                 callback(bestx)
...             if maxfev is not None and funcalls >= maxfev:
...                 stop = True
...                 break
...
...     return OptimizeResult(fun=besty, x=bestx, nit=niter,
...                           nfev=funcalls, success=(niter > 1))
>>> x0 = [1.35, 0.9, 0.8, 1.1, 1.2]
>>> res = minimize(rosen, x0, method=custmin, options=dict(stepsize=0.05))
>>> res.x
array([1., 1., 1., 1., 1.])
```

This will work just as well in case of univariate optimization:

```
>>> def custmin(fun, bracket, args=(), maxfev=None, stepsize=0.1,
...         maxiter=100, callback=None, **options):
...     bestx = (bracket[1] + bracket[0]) / 2.0
...     besty = fun(bestx)
...     funcalls = 1
...     niter = 0
...     improved = True
...     stop = False
...
...     while improved and not stop and niter < maxiter:
...         improved = False
...         niter += 1
...         for testx in [bestx - stepsize, bestx + stepsize]:
...             testy = fun(testx, *args)
...             funcalls += 1
...             if testy < besty:
...                 besty = testy
...                 bestx = testx
...                 improved = True
...         if callback is not None:
...             callback(bestx)
...         if maxfev is not None and funcalls >= maxfev:
...             stop = True
...             break
...
...     return OptimizeResult(fun=besty, x=bestx, nit=niter,
```

```
...                              nfev=funcalls, success=(niter > 1))
>>> def f(x):
...     return (x - 2)**2 * (x + 2)**2
>>> res = minimize_scalar(f, bracket=(-3.5, 0), method=custmin,
...                       options=dict(stepsize = 0.05))
>>> res.x
-2.0
```

Root finding

Scalar functions

If one has a single-variable equation, there are four different root finding algorithms that can be tried. Each of these algorithms requires the endpoints of an interval in which a root is expected (because the function changes signs). In general *brentq* is the best choice, but the other methods may be useful in certain circumstances or for academic purposes.

Fixed-point solving

A problem closely related to finding the zeros of a function is the problem of finding a fixed-point of a function. A fixed point of a function is the point at which evaluation of the function returns the point: $g(x) = x$. Clearly the fixed point of g is the root of $f(x) = g(x) - x$. Equivalently, the root of f is the fixed_point of $g(x) = f(x) + x$. The routine *fixed_point* provides a simple iterative method using Aitkens sequence acceleration to estimate the fixed point of g given a starting point.

Sets of equations

Finding a root of a set of non-linear equations can be achieve using the *root* function. Several methods are available, amongst which `hybr` (the default) and `lm` which respectively use the hybrid method of Powell and the Levenberg-Marquardt method from MINPACK.

The following example considers the single-variable transcendental equation x+2cos $(x) = 0, arootofwhichcanbefoundasfollows:$

```
>>> import numpy as np
>>> from scipy.optimize import root
>>> def func(x):
...     return x + 2 * np.cos(x)
>>> sol = root(func, 0.3)
>>> sol.x
array([-1.02986653])
>>> sol.fun
array([ -6.66133815e-16])
```

Consider now a set of non-linear equations

$$\begin{aligned} x_0 \cos(x_1) &= 4, \\ x_0 x_1 - x_1 &= 5. \end{aligned}$$

We define the objective function so that it also returns the Jacobian and indicate this by setting the `jac` parameter to `True`. Also, the Levenberg-Marquardt solver is used here.

```
>>> def func2(x):
...     f = [x[0] * np.cos(x[1]) - 4,
...          x[1]*x[0] - x[1] - 5]
...     df = np.array([[np.cos(x[1]), -x[0] * np.sin(x[1])],
...                    [x[1], x[0] - 1]])
```

```
...     return f, df
>>> sol = root(func2, [1, 1], jac=True, method='lm')
>>> sol.x
array([ 6.50409711,  0.90841421])
```

Root finding for large problems

Methods `hybr` and `lm` in *root* cannot deal with a very large number of variables (*N*), as they need to calculate and invert a dense *N x N* Jacobian matrix on every Newton step. This becomes rather inefficient when *N* grows.

Consider for instance the following problem: we need to solve the following integrodifferential equation on the square $[0, 1] \times [0, 1]$:

$$(\partial_x^2 + \partial_y^2)P + 5\left(\int_0^1\int_0^1 \cosh(P)\,dx\,dy\right)^2 = 0$$

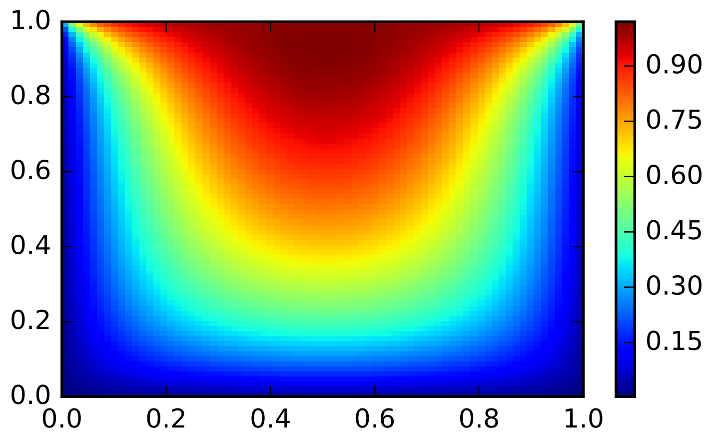
with the boundary condition $P(x, 1) = 1$ on the upper edge and $P = 0$ elsewhere on the boundary of the square. This can be done by approximating the continuous function *P* by its values on a grid, $P_{n,m} \approx P(nh, mh)$, with a small grid spacing h. The derivatives and integrals can then be approximated; for instance $\partial_x^2 P(x, y) \approx (P(x + h, y) - 2P(x, y) + P(x - h, y))/h^2$. The problem is then equivalent to finding the root of some function `residual(P)`, where `P` is a vector of length $N_x N_y$.

Now, because $N_x N_y$ can be large, methods `hybr` or `lm` in *root* will take a long time to solve this problem. The solution can however be found using one of the large-scale solvers, for example `krylov`, `broyden2`, or `anderson`. These use what is known as the inexact Newton method, which instead of computing the Jacobian matrix exactly, forms an approximation for it.

The problem we have can now be solved as follows:

```
import numpy as np
from scipy.optimize import root
from numpy import cosh, zeros_like, mgrid, zeros

# parameters
nx, ny = 75, 75
hx, hy = 1./(nx-1), 1./(ny-1)

P_left, P_right = 0, 0
P_top, P_bottom = 1, 0

def residual(P):
   d2x = zeros_like(P)
   d2y = zeros_like(P)

   d2x[1:-1] = (P[2:]    - 2*P[1:-1] + P[:-2]) / hx/hx
   d2x[0]    = (P[1]     - 2*P[0]    + P_left)/hx/hx
   d2x[-1]   = (P_right - 2*P[-1]    + P[-2])/hx/hx

   d2y[:,1:-1] = (P[:,2:] - 2*P[:,1:-1] + P[:,:-2])/hy/hy
   d2y[:,0]    = (P[:,1]  - 2*P[:,0]    + P_bottom)/hy/hy
   d2y[:,-1]   = (P_top   - 2*P[:,-1]   + P[:,-2])/hy/hy

   return d2x + d2y + 5*cosh(P).mean()**2

# solve
guess = zeros((nx, ny), float)
sol = root(residual, guess, method='krylov', options={'disp': True})
#sol = root(residual, guess, method='broyden2', options={'disp': True, 'max_rank': 50}
↪)
```

```
#sol = root(residual, guess, method='anderson', options={'disp': True, 'M': 10})
print('Residual: %g' % abs(residual(sol.x)).max())

# visualize
import matplotlib.pyplot as plt
x, y = mgrid[0:1:(nx*1j), 0:1:(ny*1j)]
plt.pcolor(x, y, sol.x)
plt.colorbar()
plt.show()
```



Still too slow? Preconditioning.

When looking for the zero of the functions $f_i(\mathbf{x}) = 0$, *i = 1, 2, ..., N*, the `krylov` solver spends most of its time inverting the Jacobian matrix,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

If you have an approximation for the inverse matrix $M \approx J^{-1}$, you can use it for *preconditioning* the linear inversion problem. The idea is that instead of solving $J\mathbf{s} = \mathbf{y}$ one solves $MJ\mathbf{s} = M\mathbf{y}$: since matrix MJ is "closer" to the identity matrix than J is, the equation should be easier for the Krylov method to deal with.

The matrix *M* can be passed to *root* with method `krylov` as an option `options['jac_options']['inner_M']`. It can be a (sparse) matrix or a *scipy.sparse.linalg.LinearOperator* instance.

For the problem in the previous section, we note that the function to solve consists of two parts: the first one is application of the Laplace operator, $[\partial_x^2 + \partial_y^2]P$, and the second is the integral. We can actually easily compute the Jacobian corresponding to the Laplace operator part: we know that in one dimension

$$\partial_x^2 \approx \frac{1}{h_x^2} \begin{pmatrix} -2 & 1 & 0 & 0 \cdots \\ 1 & -2 & 1 & 0 \cdots \\ 0 & 1 & -2 & 1 \cdots \\ \ldots \end{pmatrix} = h_x^{-2} L$$

so that the whole 2-D operator is represented by

$$J_1 = \partial_x^2 + \partial_y^2 \simeq h_x^{-2} L \otimes I + h_y^{-2} I \otimes L$$

The matrix J_2 of the Jacobian corresponding to the integral is more difficult to calculate, and since *all* of it entries are nonzero, it will be difficult to invert. J_1 on the other hand is a relatively simple matrix, and can be inverted by `scipy.sparse.linalg.splu` (or the inverse can be approximated by `scipy.sparse.linalg.spilu`). So we are content to take $M \approx J_1^{-1}$ and hope for the best.

In the example below, we use the preconditioner $M = J_1^{-1}$.

```
import numpy as np
from scipy.optimize import root
from scipy.sparse import spdiags, kron
from scipy.sparse.linalg import spilu, LinearOperator
from numpy import cosh, zeros_like, mgrid, zeros, eye

# parameters
nx, ny = 75, 75
hx, hy = 1./(nx-1), 1./(ny-1)

P_left, P_right = 0, 0
P_top, P_bottom = 1, 0

def get_preconditioner():
    """Compute the preconditioner M"""
    diags_x = zeros((3, nx))
    diags_x[0,:] = 1/hx/hx
    diags_x[1,:] = -2/hx/hx
    diags_x[2,:] = 1/hx/hx
    Lx = spdiags(diags_x, [-1,0,1], nx, nx)

    diags_y = zeros((3, ny))
    diags_y[0,:] = 1/hy/hy
    diags_y[1,:] = -2/hy/hy
    diags_y[2,:] = 1/hy/hy
    Ly = spdiags(diags_y, [-1,0,1], ny, ny)

    J1 = kron(Lx, eye(ny)) + kron(eye(nx), Ly)

    # Now we have the matrix `J_1`. We need to find its inverse `M` --
    # however, since an approximate inverse is enough, we can use
    # the *incomplete LU* decomposition

    J1_ilu = spilu(J1)

    # This returns an object with a method .solve() that evaluates
    # the corresponding matrix-vector product. We need to wrap it into
    # a LinearOperator before it can be passed to the Krylov methods:

    M = LinearOperator(shape=(nx*ny, nx*ny), matvec=J1_ilu.solve)
    return M

def solve(preconditioning=True):
    """Compute the solution"""
    count = [0]

    def residual(P):
        count[0] += 1

        d2x = zeros_like(P)
        d2y = zeros_like(P)
```

```
        d2x[1:-1] = (P[2:]    - 2*P[1:-1] + P[:-2])/hx/hx
        d2x[0]    = (P[1]     - 2*P[0]    + P_left)/hx/hx
        d2x[-1]   = (P_right - 2*P[-1]    + P[-2])/hx/hx

        d2y[:,1:-1] = (P[:,2:] - 2*P[:,1:-1] + P[:,:-2])/hy/hy
        d2y[:,0]    = (P[:,1]  - 2*P[:,0]    + P_bottom)/hy/hy
        d2y[:,-1]   = (P_top   - 2*P[:,-1]   + P[:,-2])/hy/hy

        return d2x + d2y + 5*cosh(P).mean()**2

    # preconditioner
    if preconditioning:
        M = get_preconditioner()
    else:
        M = None

    # solve
    guess = zeros((nx, ny), float)

    sol = root(residual, guess, method='krylov',
               options={'disp': True,
                        'jac_options': {'inner_M': M}})
    print 'Residual', abs(residual(sol.x)).max()
    print 'Evaluations', count[0]

    return sol.x

def main():
    sol = solve(preconditioning=True)

    # visualize
    import matplotlib.pyplot as plt
    x, y = mgrid[0:1:(nx*1j), 0:1:(ny*1j)]
    plt.clf()
    plt.pcolor(x, y, sol)
    plt.clim(0, 1)
    plt.colorbar()
    plt.show()

if __name__ == "__main__":
    main()
```

Resulting run, first without preconditioning:

```
0:   |F(x)| = 803.614; step 1; tol 0.000257947
1:   |F(x)| = 345.912; step 1; tol 0.166755
2:   |F(x)| = 139.159; step 1; tol 0.145657
3:   |F(x)| = 27.3682; step 1; tol 0.0348109
4:   |F(x)| = 1.03303; step 1; tol 0.00128227
5:   |F(x)| = 0.0406634; step 1; tol 0.00139451
6:   |F(x)| = 0.00344341; step 1; tol 0.00645373
7:   |F(x)| = 0.000153671; step 1; tol 0.00179246
8:   |F(x)| = 6.7424e-06; step 1; tol 0.00173256
Residual 3.57078908664e-07
Evaluations 317
```

and then with preconditioning:

```
0:  |F(x)| = 136.993; step 1; tol 7.49599e-06
1:  |F(x)| = 4.80983; step 1; tol 0.00110945
2:  |F(x)| = 0.195942; step 1; tol 0.00149362
3:  |F(x)| = 0.000563597; step 1; tol 7.44604e-06
4:  |F(x)| = 1.00698e-09; step 1; tol 2.87308e-12
Residual 9.29603061195e-11
Evaluations 77
```

Using a preconditioner reduced the number of evaluations of the `residual` function by a factor of *4*. For problems where the residual is expensive to compute, good preconditioning can be crucial — it can even decide whether the problem is solvable in practice or not.

Preconditioning is an art, science, and industry. Here, we were lucky in making a simple choice that worked reasonably well, but there is a lot more depth to this topic than is shown here.

References

Some further reading and related software:

3.1.6 Interpolation (`scipy.interpolate`)

Contents

- *Interpolation (`scipy.interpolate`)*
  - *1-D interpolation (`interp1d`)*
  - *Multivariate data interpolation (`griddata`)*
  - *Spline interpolation*
    - *Spline interpolation in 1-d: Procedural (interpolate.splXXX)*
    - *Spline interpolation in 1-d: Object-oriented (`UnivariateSpline`)*
    - *Two-dimensional spline representation: Procedural (`bisplrep`)*
    - *Two-dimensional spline representation: Object-oriented (`BivariateSpline`)*
  - *Using radial basis functions for smoothing/interpolation*
    - *1-d Example*
    - *2-d Example*

There are several general interpolation facilities available in SciPy, for data in 1, 2, and higher dimensions:

- A class representing an interpolant (`interp1d`) in 1-D, offering several interpolation methods.
- Convenience function `griddata` offering a simple interface to interpolation in N dimensions (N = 1, 2, 3, 4, ...). Object-oriented interface for the underlying routines is also available.
- Functions for 1- and 2-dimensional (smoothed) cubic-spline interpolation, based on the FORTRAN library FITPACK. There are both procedural and object-oriented interfaces for the FITPACK library.
- Interpolation using Radial Basis Functions.

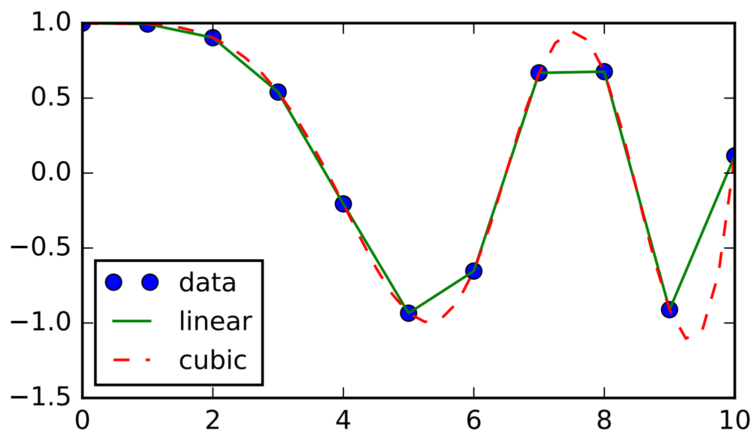
1-D interpolation (`interp1d`)

The *interp1d* class in *scipy.interpolate* is a convenient method to create a function based on fixed data points which can be evaluated anywhere within the domain defined by the given data using linear interpolation. An instance of this class is created by passing the 1-d vectors comprising the data. The instance of this class defines a __call__ method and can therefore by treated like a function which interpolates between known data values to obtain unknown values (it also has a docstring for help). Behavior at the boundary can be specified at instantiation time. The following example demonstrates its use, for linear and cubic spline interpolation:

```
>>> from scipy.interpolate import interp1d
```

```
>>> x = np.linspace(0, 10, num=11, endpoint=True)
>>> y = np.cos(-x**2/9.0)
>>> f = interp1d(x, y)
>>> f2 = interp1d(x, y, kind='cubic')
```

```
>>> xnew = np.linspace(0, 10, num=41, endpoint=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
>>> plt.legend(['data', 'linear', 'cubic'], loc='best')
>>> plt.show()
```



Multivariate data interpolation (`griddata`)

Suppose you have multidimensional data, for instance for an underlying function *f(x, y)* you only know the values at points *(x[i], y[i])* that do not form a regular grid.

Suppose we want to interpolate the 2-D function

```
>>> def func(x, y):
...     return x*(1-x)*np.cos(4*np.pi*x) * np.sin(4*np.pi*y**2)**2
```

on a grid in [0, 1]x[0, 1]

```
>>> grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]
```

but we only know its values at 1000 data points:

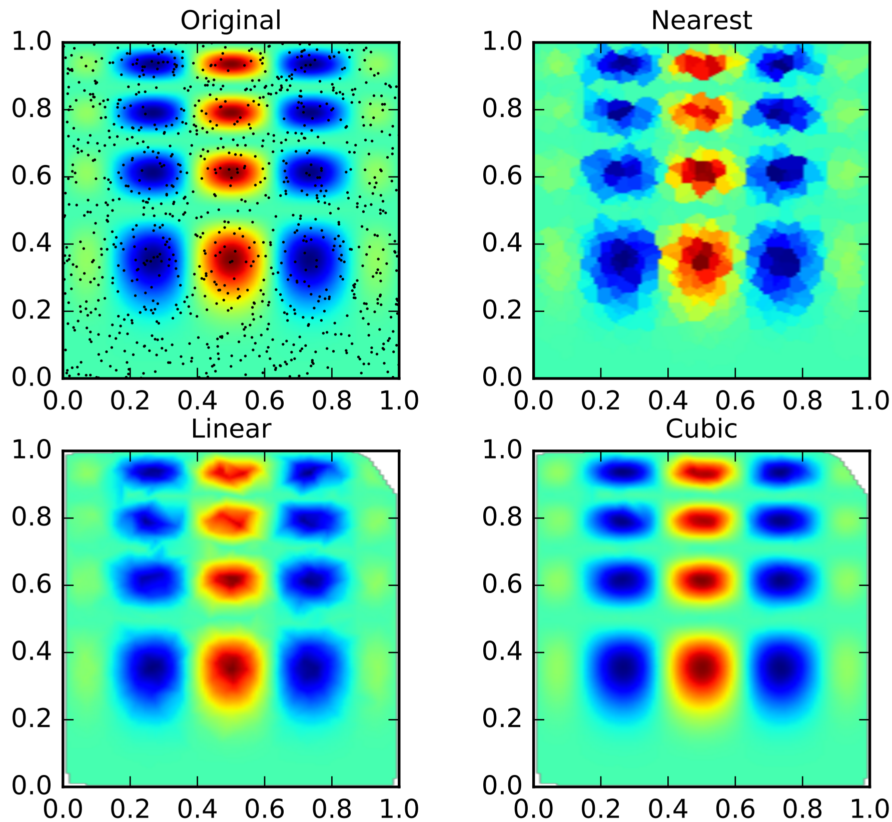
```
>>> points = np.random.rand(1000, 2)
>>> values = func(points[:,0], points[:,1])
```

This can be done with `griddata` – below we try out all of the interpolation methods:

```
>>> from scipy.interpolate import griddata
>>> grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
>>> grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
>>> grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
```

One can see that the exact result is reproduced by all of the methods to some degree, but for this smooth function the piecewise cubic interpolant gives the best results:

```
>>> import matplotlib.pyplot as plt
>>> plt.subplot(221)
>>> plt.imshow(func(grid_x, grid_y).T, extent=(0,1,0,1), origin='lower')
>>> plt.plot(points[:,0], points[:,1], 'k.', ms=1)
>>> plt.title('Original')
>>> plt.subplot(222)
>>> plt.imshow(grid_z0.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Nearest')
>>> plt.subplot(223)
>>> plt.imshow(grid_z1.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Linear')
>>> plt.subplot(224)
>>> plt.imshow(grid_z2.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Cubic')
>>> plt.gcf().set_size_inches(6, 6)
>>> plt.show()
```



Spline interpolation

Spline interpolation in 1-d: Procedural (interpolate.splXXX)

Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points. In order to find the spline representation, there are two different ways to represent a curve and obtain (smoothing) spline coefficients: directly and parametrically. The direct method finds the spline representation of a curve in a two- dimensional plane using the function `splrep`. The first two arguments are the only ones required, and these provide the x and y components of the curve. The normal output is a 3-tuple, (t, c, k) , containing the knot-points, t , the coefficients c and the order k of the spline. The default spline order is cubic, but this can be changed with the input keyword, *k.*

For curves in N -dimensional space the function `splprep` allows defining the curve parametrically. For this function only 1 input argument is required. This input is a list of N -arrays representing the curve in N -dimensional space. The length of each array is the number of curve points, and each array provides one component of the N -dimensional data point. The parameter variable is given with the keyword argument, *u,* which defaults to an equally-spaced monotonic sequence between 0 and 1 . The default output consists of two objects: a 3-tuple, (t, c, k) , containing the spline

representation and the parameter variable u.

The keyword argument, s , is used to specify the amount of smoothing to perform during the spline fit. The default value of s is $s = m - \sqrt{2m}$ where m is the number of data-points being fit. Therefore, **if no smoothing is desired a value of s** $= 0$ **should be passed to the routines.**

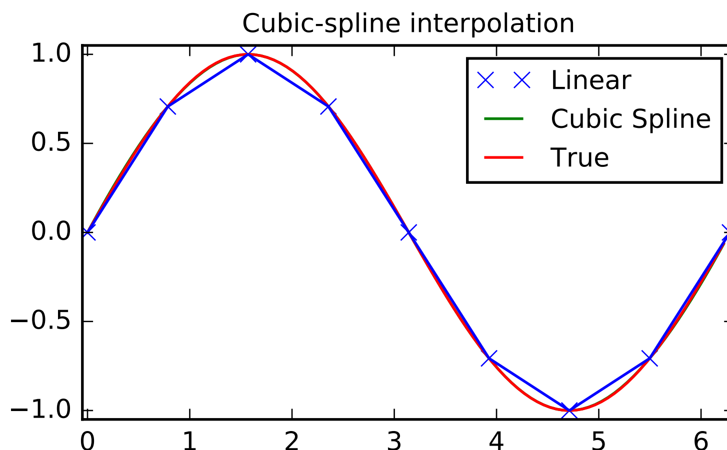
Once the spline representation of the data has been determined, functions are available for evaluating the spline (*splev*) and its derivatives (*splev*, *spalde*) at any point and the integral of the spline between any two points (*splint*). In addition, for cubic splines ($k = 3$) with 8 or more knots, the roots of the spline can be estimated (*sproot*). These functions are demonstrated in the example that follows.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
```

Cubic-spline

```
>>> x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
>>> y = np.sin(x)
>>> tck = interpolate.splrep(x, y, s=0)
>>> xnew = np.arange(0, 2*np.pi, np.pi/50)
>>> ynew = interpolate.splev(xnew, tck, der=0)
```

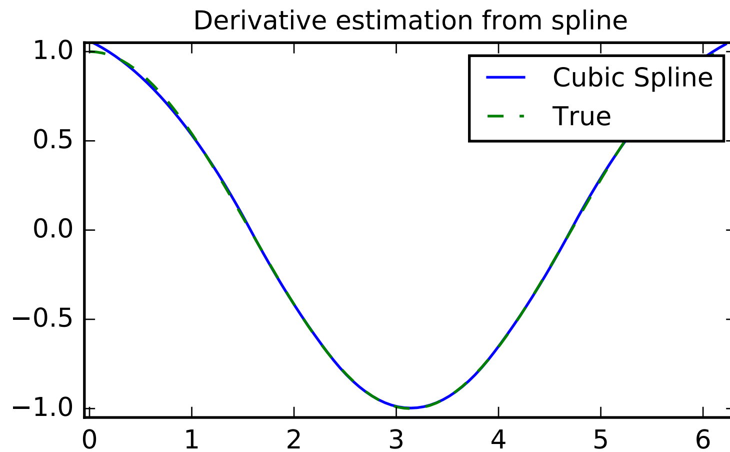
```
>>> plt.figure()
>>> plt.plot(x, y, 'x', xnew, ynew, xnew, np.sin(xnew), x, y, 'b')
>>> plt.legend(['Linear', 'Cubic Spline', 'True'])
>>> plt.axis([-0.05, 6.33, -1.05, 1.05])
>>> plt.title('Cubic-spline interpolation')
>>> plt.show()
```



Derivative of spline

```
>>> yder = interpolate.splev(xnew, tck, der=1)
>>> plt.figure()
>>> plt.plot(xnew, yder, xnew, np.cos(xnew),'--')
>>> plt.legend(['Cubic Spline', 'True'])
>>> plt.axis([-0.05, 6.33, -1.05, 1.05])
```

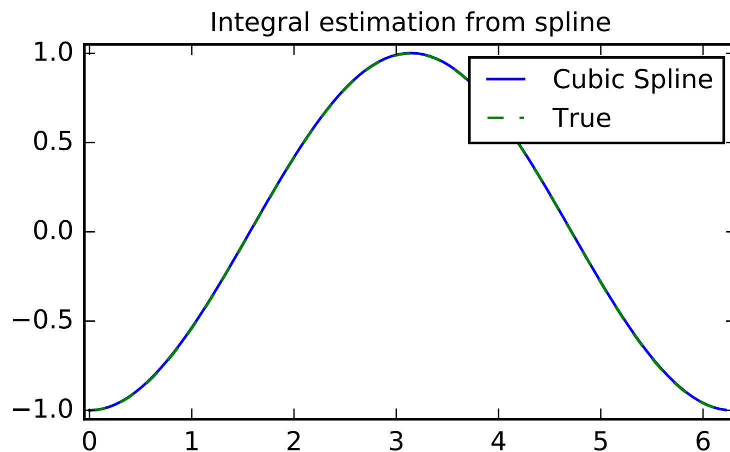
```
>>> plt.title('Derivative estimation from spline')
>>> plt.show()
```



Integral of spline

```
>>> def integ(x, tck, constant=-1):
...     x = np.atleast_1d(x)
...     out = np.zeros(x.shape, dtype=x.dtype)
...     for n in range(len(out)):
...         out[n] = interpolate.splint(0, x[n], tck)
...     out += constant
...     return out
```

```
>>> yint = integ(xnew, tck)
>>> plt.figure()
>>> plt.plot(xnew, yint, xnew, -np.cos(xnew), '--')
>>> plt.legend(['Cubic Spline', 'True'])
>>> plt.axis([-0.05, 6.33, -1.05, 1.05])
>>> plt.title('Integral estimation from spline')
>>> plt.show()
```



Roots of spline

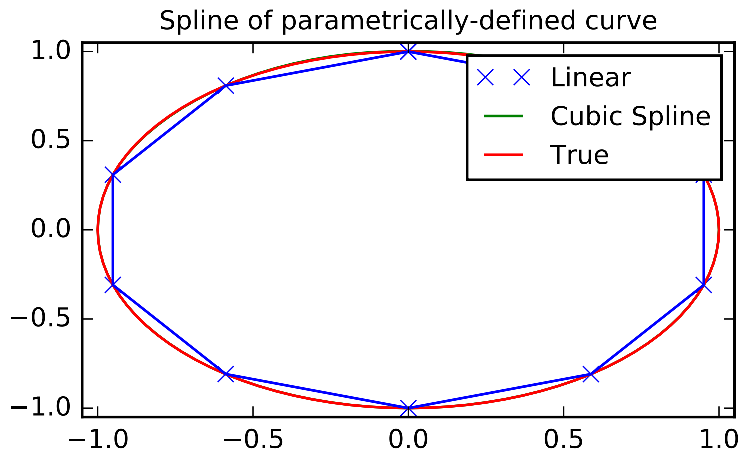
```
>>> interpolate.sproot(tck)
array([3.1416])
```

Notice that *sproot* failed to find an obvious solution at the edge of the approximation interval, $x = 0$. If we define the spline on a slightly larger interval, we recover both roots $x = 0$ and $x = 2\pi$:

```
>>> x = np.linspace(-np.pi/4, 2.*np.pi + np.pi/4, 21)
>>> y = np.sin(x)
>>> tck = interpolate.splrep(x, y, s=0)
>>> interpolate.sproot(tck)
array([0., 3.1416])
```

Parametric spline

```
>>> t = np.arange(0, 1.1, .1)
>>> x = np.sin(2*np.pi*t)
>>> y = np.cos(2*np.pi*t)
>>> tck, u = interpolate.splprep([x, y], s=0)
>>> unew = np.arange(0, 1.01, 0.01)
>>> out = interpolate.splev(unew, tck)
>>> plt.figure()
>>> plt.plot(x, y, 'x', out[0], out[1], np.sin(2*np.pi*unew), np.cos(2*np.pi*unew), x,
↪ y, 'b')
>>> plt.legend(['Linear', 'Cubic Spline', 'True'])
>>> plt.axis([-1.05, 1.05, -1.05, 1.05])
>>> plt.title('Spline of parametrically-defined curve')
>>> plt.show()
```



Spline interpolation in 1-d: Object-oriented (`UnivariateSpline`)

The spline-fitting capabilities described above are also available via an objected-oriented interface. The one dimensional splines are objects of the `UnivariateSpline` class, and are created with the x and y components of the curve provided as arguments to the constructor. The class defines `__call__`, allowing the object to be called with the x-axis values at which the spline should be evaluated, returning the interpolated y-values. This is shown in the example below for the subclass `InterpolatedUnivariateSpline`. The `integral`, `derivatives`, and `roots` methods are also available on `UnivariateSpline` objects, allowing definite integrals, derivatives, and roots to be computed for the spline.

The UnivariateSpline class can also be used to smooth data by providing a non-zero value of the smoothing parameter `s`, with the same meaning as the `s` keyword of the *splrep* function described above. This results in a spline that has fewer knots than the number of data points, and hence is no longer strictly an interpolating spline, but rather a smoothing spline. If this is not desired, the `InterpolatedUnivariateSpline` class is available. It is a subclass of `UnivariateSpline` that always passes through all points (equivalent to forcing the smoothing parameter to 0). This class is demonstrated in the example below.

The `LSQUnivariateSpline` class is the other subclass of `UnivariateSpline`. It allows the user to specify the number and location of internal knots explicitly with the parameter *t*. This allows creation of customized splines with non-linear spacing, to interpolate in some domains and smooth in others, or change the character of the spline.

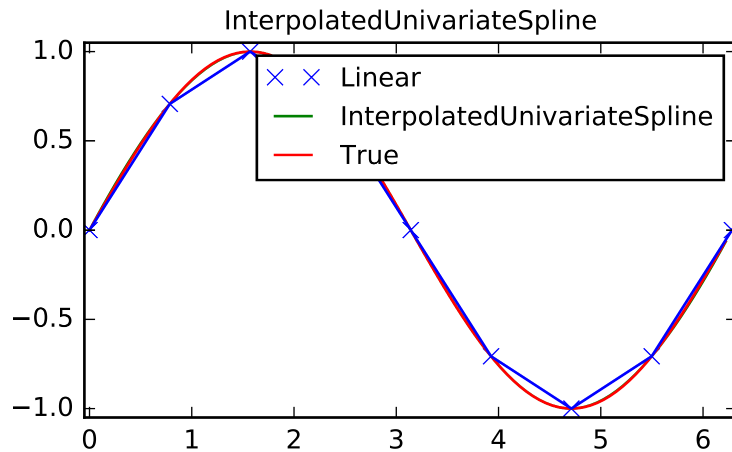
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
```

InterpolatedUnivariateSpline

```
>>> x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
>>> y = np.sin(x)
>>> s = interpolate.InterpolatedUnivariateSpline(x, y)
>>> xnew = np.arange(0, 2*np.pi, np.pi/50)
>>> ynew = s(xnew)
```

```
>>> plt.figure()
>>> plt.plot(x, y, 'x', xnew, ynew, xnew, np.sin(xnew), x, y, 'b')
>>> plt.legend(['Linear', 'InterpolatedUnivariateSpline', 'True'])
>>> plt.axis([-0.05, 6.33, -1.05, 1.05])
```

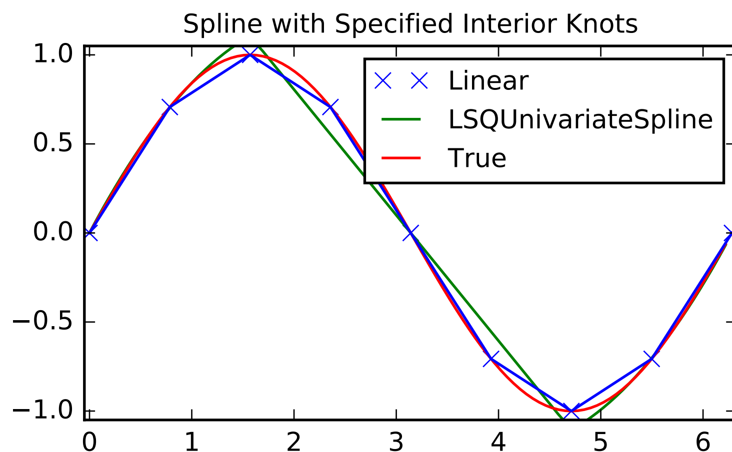
```
>>> plt.title('InterpolatedUnivariateSpline')
>>> plt.show()
```



LSQUnivarateSpline with non-uniform knots

```
>>> t = [np.pi/2-.1, np.pi/2+.1, 3*np.pi/2-.1, 3*np.pi/2+.1]
>>> s = interpolate.LSQUnivariateSpline(x, y, t, k=2)
>>> ynew = s(xnew)
```

```
>>> plt.figure()
>>> plt.plot(x, y, 'x', xnew, ynew, xnew, np.sin(xnew), x, y, 'b')
>>> plt.legend(['Linear', 'LSQUnivariateSpline', 'True'])
>>> plt.axis([-0.05, 6.33, -1.05, 1.05])
>>> plt.title('Spline with Specified Interior Knots')
>>> plt.show()
```



Two-dimensional spline representation: Procedural (`bisplrep`)

For (smooth) spline-fitting to a two dimensional surface, the function *bisplrep* is available. This function takes as required inputs the **1-D** arrays *x*, *y*, and *z* which represent points on the surface $z = f(x, y)$. The default output is a list $[tx, ty, c, kx, ky]$ whose entries represent respectively, the components of the knot positions, the coefficients of the spline, and the order of the spline in each coordinate. It is convenient to hold this list in a single object, *tck,* so that it can be passed easily to the function *bisplev*. The keyword, s, can be used to change the amount of smoothing performed on the data while determining the appropriate spline. The default value is $s = m - \sqrt{2m}$ where m is the number of data points in the *x, y,* and *z* vectors. As a result, if no smoothing is desired, then $s = 0$ should be passed to *bisplrep*.

To evaluate the two-dimensional spline and it's partial derivatives (up to the order of the spline), the function *bisplev* is required. This function takes as the first two arguments **two 1-D arrays** whose cross-product specifies the domain over which to evaluate the spline. The third argument is the *tck* list returned from *bisplrep*. If desired, the fourth and fifth arguments provide the orders of the partial derivative in the x and y direction respectively.

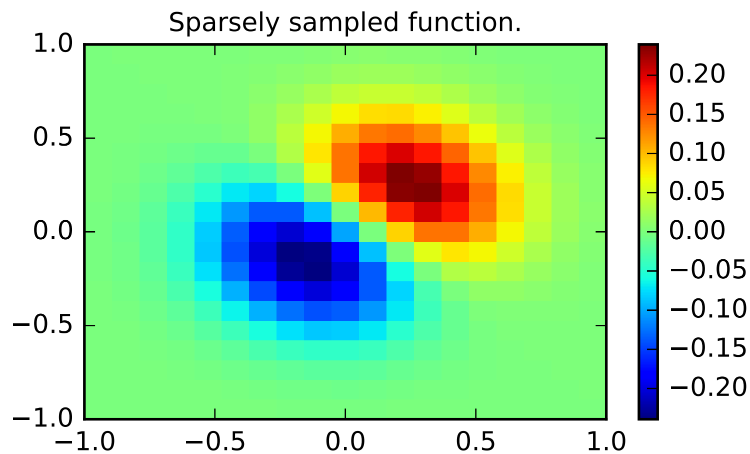
It is important to note that two dimensional interpolation should not be used to find the spline representation of images. The algorithm used is not amenable to large numbers of input points. The signal processing toolbox contains more appropriate algorithms for finding the spline representation of an image. The two dimensional interpolation commands are intended for use when interpolating a two dimensional function as shown in the example that follows. This example uses the `mgrid` command in NumPy which is useful for defining a "mesh-grid" in many dimensions. (See also the `ogrid` command if the full-mesh is not needed). The number of output arguments and the number of dimensions of each argument is determined by the number of indexing objects passed in `mgrid`.

```
>>> import numpy as np
>>> from scipy import interpolate
>>> import matplotlib.pyplot as plt
```

Define function over sparse 20x20 grid

```
>>> x, y = np.mgrid[-1:1:20j, -1:1:20j]
>>> z = (x+y) * np.exp(-6.0*(x*x+y*y))
```

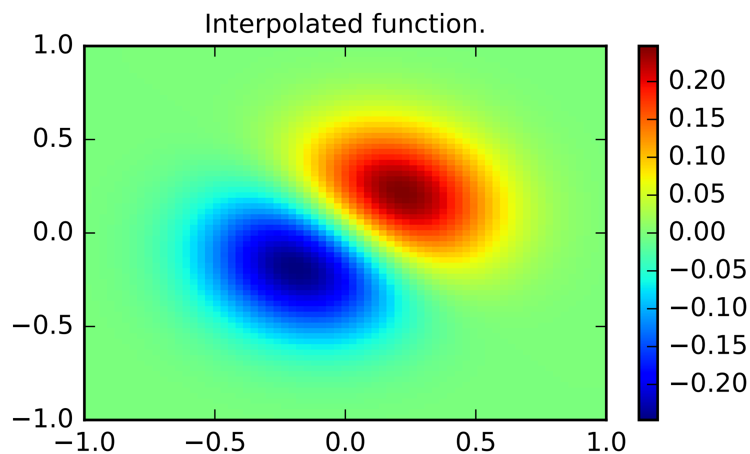
```
>>> plt.figure()
>>> plt.pcolor(x, y, z)
>>> plt.colorbar()
>>> plt.title("Sparsely sampled function.")
>>> plt.show()
```



Interpolate function over new 70x70 grid

```
>>> xnew, ynew = np.mgrid[-1:1:70j, -1:1:70j]
>>> tck = interpolate.bisplrep(x, y, z, s=0)
>>> znew = interpolate.bisplev(xnew[:,0], ynew[0,:], tck)
```

```
>>> plt.figure()
>>> plt.pcolor(xnew, ynew, znew)
>>> plt.colorbar()
>>> plt.title("Interpolated function.")
>>> plt.show()
```



Two-dimensional spline representation: Object-oriented (`BivariateSpline`)

The `BivariateSpline` class is the 2-dimensional analog of the `UnivariateSpline` class. It and its subclasses implement the FITPACK functions described above in an object oriented fashion, allowing objects to be instantiated that can be called to compute the spline value by passing in the two coordinates as the two arguments.

Using radial basis functions for smoothing/interpolation

Radial basis functions can be used for smoothing/interpolating scattered data in n-dimensions, but should be used with caution for extrapolation outside of the observed data range.

1-d Example

This example compares the usage of the `Rbf` and `UnivariateSpline` classes from the scipy.interpolate module.

```
>>> import numpy as np
>>> from scipy.interpolate import Rbf, InterpolatedUnivariateSpline
>>> import matplotlib.pyplot as plt
```

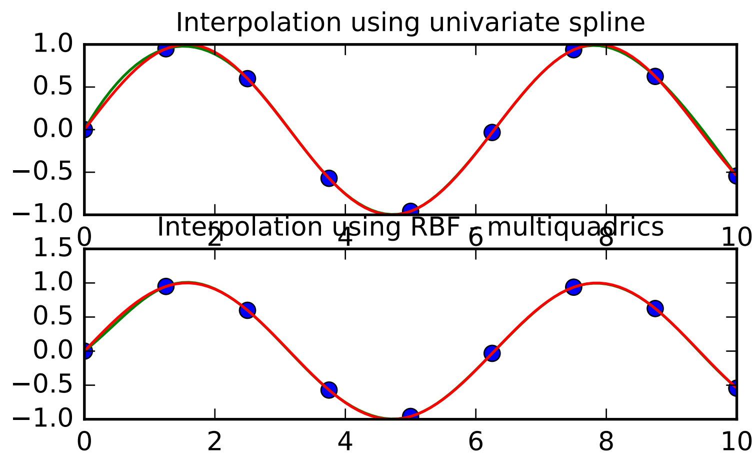
```
>>> # setup data
>>> x = np.linspace(0, 10, 9)
>>> y = np.sin(x)
>>> xi = np.linspace(0, 10, 101)
```

```
>>> # use fitpack2 method
>>> ius = InterpolatedUnivariateSpline(x, y)
>>> yi = ius(xi)
```

```
>>> plt.subplot(2, 1, 1)
>>> plt.plot(x, y, 'bo')
>>> plt.plot(xi, yi, 'g')
>>> plt.plot(xi, np.sin(xi), 'r')
>>> plt.title('Interpolation using univariate spline')
```

```
>>> # use RBF method
>>> rbf = Rbf(x, y)
>>> fi = rbf(xi)
```

```
>>> plt.subplot(2, 1, 2)
>>> plt.plot(x, y, 'bo')
>>> plt.plot(xi, fi, 'g')
>>> plt.plot(xi, np.sin(xi), 'r')
>>> plt.title('Interpolation using RBF - multiquadrics')
>>> plt.show()
```



2-d Example

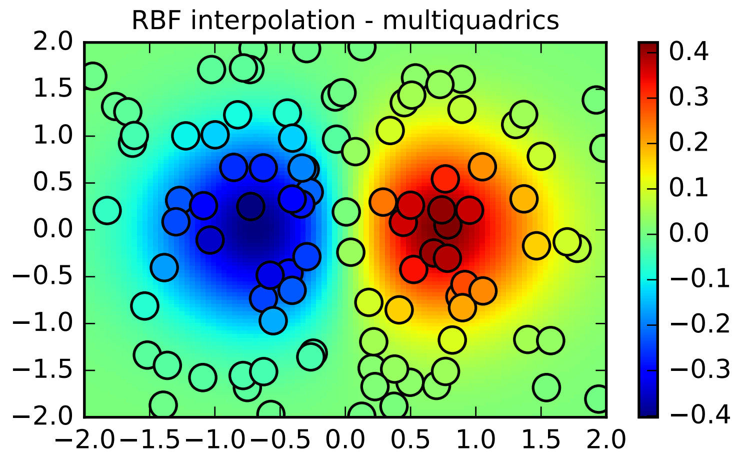
This example shows how to interpolate scattered 2d data.

```
>>> import numpy as np
>>> from scipy.interpolate import Rbf
>>> import matplotlib.pyplot as plt
>>> from matplotlib import cm
```

```
>>> # 2-d tests - setup scattered data
>>> x = np.random.rand(100)*4.0-2.0
>>> y = np.random.rand(100)*4.0-2.0
>>> z = x*np.exp(-x**2-y**2)
>>> ti = np.linspace(-2.0, 2.0, 100)
>>> XI, YI = np.meshgrid(ti, ti)
```

```
>>> # use RBF
>>> rbf = Rbf(x, y, z, epsilon=2)
>>> ZI = rbf(XI, YI)
```

```
>>> # plot the result
>>> plt.subplot(1, 1, 1)
>>> plt.pcolor(XI, YI, ZI, cmap=cm.jet)
>>> plt.scatter(x, y, 100, z, cmap=cm.jet)
>>> plt.title('RBF interpolation - multiquadrics')
>>> plt.xlim(-2, 2)
>>> plt.ylim(-2, 2)
>>> plt.colorbar()
```



3.1.7 Fourier Transforms (`scipy.fftpack`)

Contents

- *Fourier Transforms* (`scipy.fftpack`)
  - *Fast Fourier transforms*
    - *One dimensional discrete Fourier transforms*
    - *Two and n-dimensional discrete Fourier transforms*
    - *FFT convolution*
  - *Discrete Cosine Transforms*
    - *Type I DCT*
    - *Type II DCT*
    - *Type III DCT*
    - *DCT and IDCT*
    - *Example*
  - *Discrete Sine Transforms*
    - *Type I DST*
    - *Type II DST*
    - *Type III DST*
    - *DST and IDST*
  - *Cache Destruction*
  - *References*

Fourier analysis is a method for expressing a function as a sum of periodic components, and for recovering the signal

from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT). The DFT has become a mainstay of numerical computing in part because of a very fast algorithm for computing it, called the Fast Fourier Transform (FFT), which was known to Gauss (1805) and was brought to light in its current form by Cooley and Tukey *[CT65]*. Press et al. *[NR07]* provide an accessible introduction to Fourier analysis and its applications.

Note: PyFFTW provides a way to replace a number of functions in `scipy.fftpack` with its own functions, which are usually significantly faster, via pyfftw.interfaces. Because PyFFTW relies on the GPL-licensed FFTW it cannot be included in Scipy. Users for whom the speed of FFT routines is critical should consider installing PyFFTW.

Fast Fourier transforms

One dimensional discrete Fourier transforms

The FFT *y[k]* of length N of the length-N sequence *x[n]* is defined as

$$y[k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} x[n] \,,$$

and the inverse transform is defined as follows

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi j \frac{kn}{N}} y[k] \,.$$

These transforms can be calculated by means of `fft` and `ifft`, respectively as shown in the following example.

```
>>> from scipy.fftpack import fft, ifft
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>> y = fft(x)
>>> y
array([ 4.50000000+0.j        ,  2.08155948-1.65109876j,
       -1.83155948+1.60822041j, -1.83155948-1.60822041j,
        2.08155948+1.65109876j])
>>> yinv = ifft(y)
>>> yinv
array([ 1.0+0.j,  2.0+0.j,  1.0+0.j, -1.0+0.j,  1.5+0.j])
```

From the definition of the FFT it can be seen that

$$y[0] = \sum_{n=0}^{N-1} x[n] \,.$$

In the example

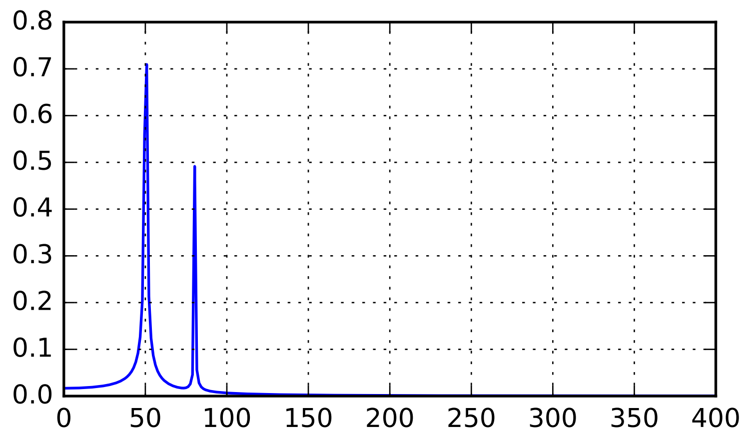
```
>>> np.sum(x)
4.5
```

which corresponds to $y[0]$. For N even, the elements $y[1] ... y[N/2-1]$ contain the positive-frequency terms, and the elements $y[N/2] ... y[N-1]$ contain the negative-frequency terms, in order of decreasingly negative frequency. For N odd, the elements $y[1] ... y[(N-1)/2]$ contain the positive- frequency terms, and the elements $y[(N+1)/2] ... y[N-1]$ contain the negative- frequency terms, in order of decreasingly negative frequency.

In case the sequence x is real-valued, the values of $y[n]$ for positive frequencies is the conjugate of the values $y[n]$ for negative frequencies (because the spectrum is symmetric). Typically, only the FFT corresponding to positive frequencies is plotted.

The example plots the FFT of the sum of two sines.

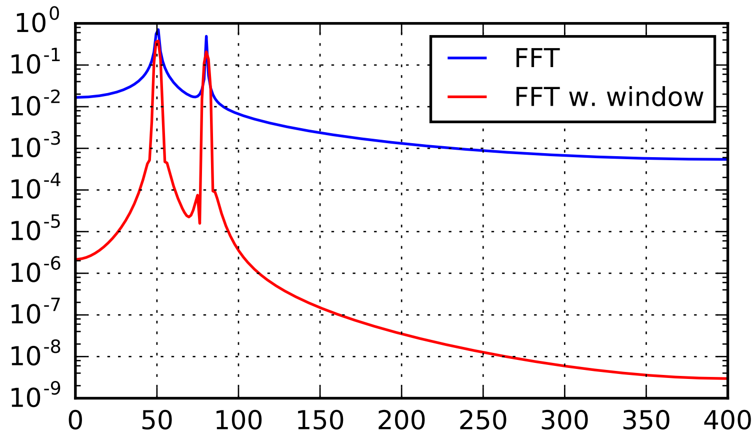
```
>>> from scipy.fftpack import fft
>>> # Number of sample points
>>> N = 600
>>> # sample spacing
>>> T = 1.0 / 800.0
>>> x = np.linspace(0.0, N*T, N)
>>> y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
>>> yf = fft(y)
>>> xf = np.linspace(0.0, 1.0/(2.0*T), N//2)
>>> import matplotlib.pyplot as plt
>>> plt.plot(xf, 2.0/N * np.abs(yf[0:N//2]))
>>> plt.grid()
>>> plt.show()
```



The FFT input signal is inherently truncated. This truncation can be modelled as multiplication of an infinite signal with a rectangular window function. In the spectral domain this multiplication becomes convolution of the signal spectrum with the window function spectrum, being of form $\sin(x)/x$. This convolution is the cause of an effect called spectral leakage (see *[WPW]*). Windowing the signal with a dedicated window function helps mitigate spectral leakage. The example below uses a Blackman window from scipy.signal and shows the effect of windowing (the zero component of the FFT has been truncated for illustrative purposes).

```
>>> from scipy.fftpack import fft
>>> # Number of sample points
>>> N = 600
>>> # sample spacing
>>> T = 1.0 / 800.0
>>> x = np.linspace(0.0, N*T, N)
>>> y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
>>> yf = fft(y)
>>> from scipy.signal import blackman
>>> w = blackman(N)
>>> ywf = fft(y*w)
>>> xf = np.linspace(0.0, 1.0/(2.0*T), N/2)
>>> import matplotlib.pyplot as plt
>>> plt.semilogy(xf[1:N//2], 2.0/N * np.abs(yf[1:N//2]), '-b')
>>> plt.semilogy(xf[1:N//2], 2.0/N * np.abs(ywf[1:N//2]), '-r')
```

```
>>> plt.legend(['FFT', 'FFT w. window'])
>>> plt.grid()
>>> plt.show()
```



In case the sequence x is complex-valued, the spectrum is no longer symmetric. To simplify working wit the FFT functions, scipy provides the following two helper functions.

The function `fftfreq` returns the FFT sample frequency points.

```
>>> from scipy.fftpack import fftfreq
>>> freq = fftfreq(8, 0.125)
>>> freq
array([ 0., 1., 2., 3., -4., -3., -2., -1.])
```

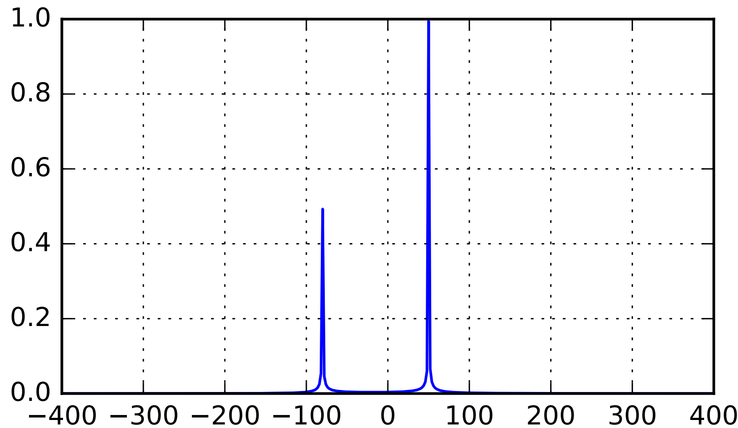
In a similar spirit, the function `fftshift` allows swapping the lower and upper halves of a vector, so that it becomes suitable for display.

```
>>> from scipy.fftpack import fftshift
>>> x = np.arange(8)
>>> fftshift(x)
array([4, 5, 6, 7, 0, 1, 2, 3])
```

The example below plots the FFT of two complex exponentials; note the asymmetric spectrum.

```
>>> from scipy.fftpack import fft, fftfreq, fftshift
>>> # number of signal points
>>> N = 400
>>> # sample spacing
>>> T = 1.0 / 800.0
>>> x = np.linspace(0.0, N*T, N)
>>> y = np.exp(50.0 * 1.j * 2.0*np.pi*x) + 0.5*np.exp(-80.0 * 1.j * 2.0*np.pi*x)
>>> yf = fft(y)
>>> xf = fftfreq(N, T)
>>> xf = fftshift(xf)
>>> yplot = fftshift(yf)
>>> import matplotlib.pyplot as plt
>>> plt.plot(xf, 1.0/N * np.abs(yplot))
```

```
>>> plt.grid()
>>> plt.show()
```



The function *rfft* calculates the FFT of a real sequence and outputs the FFT coefficients $y[n]$ with separate real and imaginary parts. In case of N being even: $[y[0], Re(y[1]), Im(y[1]), ..., Re(y[N/2])]$; in case N being odd $[y[0], Re(y[1]), Im(y[1]), ..., Re(y[N/2]), Im(y[N/2])]$.

The corresponding function *irfft* calculates the IFFT of the FFT coefficients with this special ordering.

```
>>> from scipy.fftpack import fft, rfft, irfft
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5, 1.0])
>>> fft(x)
array([ 5.50+0.j        ,  2.25-0.4330127j , -2.75-1.29903811j,
        1.50+0.j        , -2.75+1.29903811j,  2.25+0.4330127j ])
>>> yr = rfft(x)
>>> yr
array([ 5.5       ,  2.25      , -0.4330127 , -2.75      , -1.29903811,
        1.5       ])
>>> irfft(yr)
array([ 1. ,  2. ,  1. , -1. ,  1.5,  1. ])
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>> fft(x)
array([ 4.50000000+0.j        ,  2.08155948-1.65109876j,
       -1.83155948+1.60822041j, -1.83155948-1.60822041j,
        2.08155948+1.65109876j])
>>> yr = rfft(x)
>>> yr
array([ 4.5       ,  2.08155948, -1.65109876, -1.83155948,  1.60822041])
```

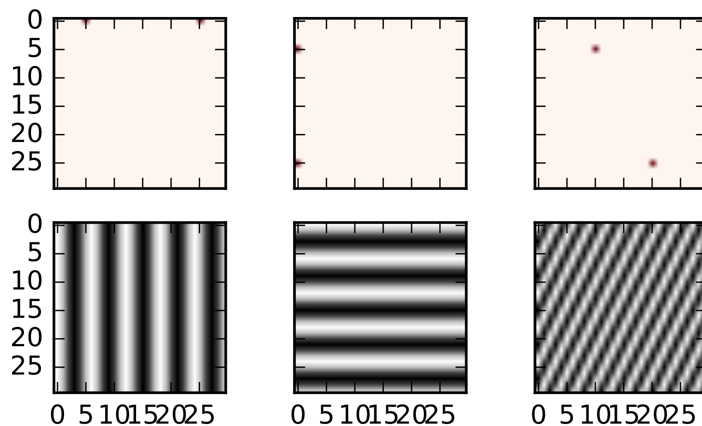
Two and n-dimensional discrete Fourier transforms

The functions *fft2* and *ifft2* provide 2-dimensional FFT, and IFFT, respectively. Similar, *fftn* and *ifftn* provide n-dimensional FFT, and IFFT, respectively.

The example below demonstrates a 2-dimensional IFFT and plots the resulting (2-dimensional) time-domain signals.

```
>>> from scipy.fftpack import ifftn
>>> import matplotlib.pyplot as plt
```

```
>>> import matplotlib.cm as cm
>>> N = 30
>>> f, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2, 3, sharex='col', sharey=
↪'row')
>>> xf = np.zeros((N,N))
>>> xf[0, 5] = 1
>>> xf[0, N-5] = 1
>>> Z = ifftn(xf)
>>> ax1.imshow(xf, cmap=cm.Reds)
>>> ax4.imshow(np.real(Z), cmap=cm.gray)
>>> xf = np.zeros((N, N))
>>> xf[5, 0] = 1
>>> xf[N-5, 0] = 1
>>> Z = ifftn(xf)
>>> ax2.imshow(xf, cmap=cm.Reds)
>>> ax5.imshow(np.real(Z), cmap=cm.gray)
>>> xf = np.zeros((N, N))
>>> xf[5, 10] = 1
>>> xf[N-5, N-10] = 1
>>> Z = ifftn(xf)
>>> ax3.imshow(xf, cmap=cm.Reds)
>>> ax6.imshow(np.real(Z), cmap=cm.gray)
>>> plt.show()
```



FFT convolution

scipy.fftpack.convolve performs a convolution of two one-dimensional arrays in frequency domain.

Discrete Cosine Transforms

Scipy provides a DCT with the function *dct* and a corresponding IDCT with the function *idct*. There are 8 types of the DCT *[WPC]*, *[Mak]*; however, only the first 3 types are implemented in scipy. “The” DCT generally refers to DCT type 2, and “the” Inverse DCT generally refers to DCT type 3. In addition, the DCT coefficients can be normalized differently (for most types, scipy provides `None` and `ortho`). Two parameters of the dct/idct function calls allow setting the DCT type and coefficient normalization.

For a single dimension array x, dct(x, norm='ortho') is equal to MATLAB dct(x).

Type I DCT

Scipy uses the following definition of the unnormalized DCT-I (`norm='None'`):

$$y[k] = x_0 + (-1)^k x_{N-1} + 2\sum_{n=1}^{N-2} x[n]\cos\left(\frac{\pi nk}{N-1}\right), \qquad 0 \le k < N.$$

Only `None` is supported as normalization mode for DCT-I. Note also that the DCT-I is only supported for input size > 1

Type II DCT

Scipy uses the following definition of the unnormalized DCT-II (`norm='None'`):

$$y[k] = 2\sum_{n=0}^{N-1} x[n]\cos\left(\frac{\pi(2n+1)k}{2N}\right) \qquad 0 \le k < N.$$

In case of the normalized DCT (`norm='ortho'`), the DCT coefficients $y[k]$ are multiplied by a scaling factor f:

$$f = \begin{cases} \sqrt{1/(4N)}, & \text{if } k = 0 \\ \sqrt{1/(2N)}, & \text{otherwise} \end{cases}.$$

In this case, the DCT "base functions" $\phi_k[n] = 2f\cos\left(\frac{\pi(2n+1)k}{2N}\right)$ become orthonormal:

$$\sum_{n=0}^{N-1} \phi_k[n]\phi_l[n] = \delta_{lk}$$

Type III DCT

Scipy uses the following definition of the unnormalized DCT-III (`norm='None'`):

$$y[k] = x_0 + 2\sum_{n=1}^{N-1} x[n]\cos\left(\frac{\pi n(2k+1)}{2N}\right) \qquad 0 \le k < N,$$

or, for `norm='ortho'`:

$$y[k] = \frac{x_0}{\sqrt{N}} + \frac{2}{\sqrt{N}}\sum_{n=1}^{N-1} x[n]\cos\left(\frac{\pi n(2k+1)}{2N}\right) \qquad 0 \le k < N.$$

DCT and IDCT

The (unnormalized) DCT-III is the inverse of the (unnormalized) DCT-II, up to a factor *2N*. The orthonormalized DCT-III is exactly the inverse of the orthonormalized DCT- II. The function *idct* performs the mappings between the DCT and IDCT types.

The example below shows the relation between DCT and IDCT for different types and normalizations.

```
>>> from scipy.fftpack import dct, idct
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>> dct(dct(x, type=2, norm='ortho'), type=3, norm='ortho')
[1.0, 2.0, 1.0, -1.0, 1.5]
>>>  # scaling factor 2*N = 10
>>> idct(dct(x, type=2), type=2)
array([ 10.,  20.,  10., -10.,  15.])
>>>  # no scaling factor
```

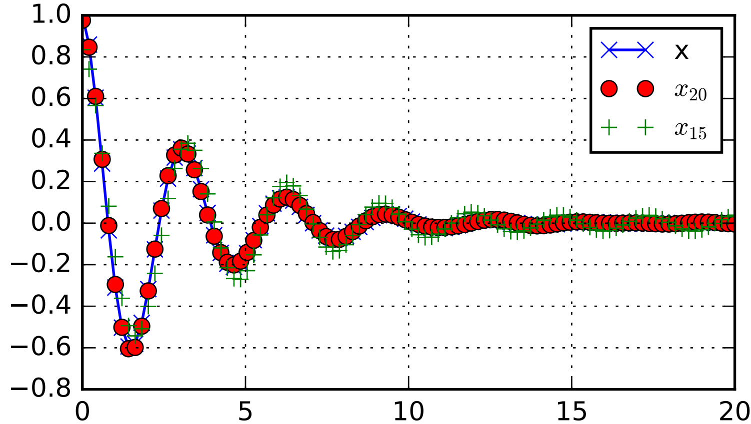
```
>>> idct(dct(x, type=2, norm='ortho'), type=2, norm='ortho')
array([ 1. ,  2. ,  1. , -1. ,  1.5])
>>>  # scaling factor 2*N = 10
>>> idct(dct(x, type=3), type=3)
array([ 10.,  20.,  10., -10.,  15.])
>>>  # no scaling factor
>>> idct(dct(x, type=3, norm='ortho'), type=3, norm='ortho')
array([ 1. ,  2. ,  1. , -1. ,  1.5])
>>>  # scaling factor 2*(N-1) = 8
>>> idct(dct(x, type=1), type=1)
array([  8.,  16.,   8.,  -8.,  12.])
```

Example

The DCT exhibits the "energy compaction property", meaning that for many signals only the first few DCT coefficients have significant magnitude. Zeroing out the other coefficients leads to a small reconstruction error, a fact which is exploited in lossy signal compression (e.g. JPEG compression).

The example below shows a signal x and two reconstructions (x_{20} and x_{15})from the signal's DCT coefficients. The signal x_{20} is reconstructed from the first 20 DCT coefficients, x_{15} is reconstructed from the first 15 DCT coefficients. It can be seen that the relative error of using 20 coefficients is still very small (~0.1%), but provides a five-fold compression rate.

```
>>> from scipy.fftpack import dct, idct
>>> import matplotlib.pyplot as plt
>>> N = 100
>>> t = np.linspace(0,20,N)
>>> x = np.exp(-t/3)*np.cos(2*t)
>>> y = dct(x, norm='ortho')
>>> window = np.zeros(N)
>>> window[:20] = 1
>>> yr = idct(y*window, norm='ortho')
>>> sum(abs(x-yr)**2) / sum(abs(x)**2)
0.0010901402257
>>> plt.plot(t, x, '-bx')
>>> plt.plot(t, yr, 'ro')
>>> window = np.zeros(N)
>>> window[:15] = 1
>>> yr = idct(y*window, norm='ortho')
>>> sum(abs(x-yr)**2) / sum(abs(x)**2)
0.0718818065008
>>> plt.plot(t, yr, 'g+')
>>> plt.legend(['x', '$x_{20}$', '$x_{15}$'])
>>> plt.grid()
>>> plt.show()
```



Discrete Sine Transforms

Scipy provides a DST *[Mak]* with the function `dst` and a corresponding IDST with the function `idst`.

There are theoretically 8 types of the DST for different combinations of even/odd boundary conditions and boundary off sets *[WPS]*, only the first 3 types are implemented in scipy.

Type I DST

DST-I assumes the input is odd around n=-1 and n=N. Scipy uses the following definition of the unnormalized DST-I (`norm='None'`):

$$y[k] = 2\sum_{n=0}^{N-1} x[n]\sin\left(\frac{\pi(n+1)(k+1)}{N+1}\right), \qquad 0 \le k < N.$$

Only `None` is supported as normalization mode for DST-I. Note also that the DST-I is only supported for input size > 1. The (unnormalized) DST-I is its own inverse, up to a factor *2(N+1)*.

Type II DST

DST-II assumes the input is odd around n=-1/2 and even around n=N. Scipy uses the following definition of the unnormalized DST-II (`norm='None'`):

$$y[k] = 2\sum_{n=0}^{N-1} x[n]\sin\left(\frac{\pi(n+1/2)(k+1)}{N}\right), \qquad 0 \le k < N.$$

Type III DST

DST-III assumes the input is odd around n=-1 and even around n=N-1. Scipy uses the following definition of the unnormalized DST-III (`norm='None'`):

$$y[k] = (-1)^k x[N-1] + 2\sum_{n=0}^{N-2} x[n]\sin\left(\frac{\pi(n+1)(k+1/2)}{N}\right), \qquad 0 \le k < N.$$

DST and IDST

The example below shows the relation between DST and IDST for different types and normalizations.

```
>>> from scipy.fftpack import dst, idst
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>>  # scaling factor 2*N = 10
>>> idst(dst(x, type=2), type=2)
array([ 10.,  20.,  10., -10.,  15.])
>>>  # no scaling factor
>>> idst(dst(x, type=2, norm='ortho'), type=2, norm='ortho')
array([ 1. ,  2. ,  1. , -1. ,  1.5])
>>>  # scaling factor 2*N = 10
>>> idst(dst(x, type=3), type=3)
array([ 10.,  20.,  10., -10.,  15.])
>>>  # no scaling factor
>>> idst(dst(x, type=3, norm='ortho'), type=3, norm='ortho')
array([ 1. ,  2. ,  1. , -1. ,  1.5])
>>>  # scaling factor 2*(N+1) = 8
>>> idst(dst(x, type=1), type=1)
array([ 12.,  24.,  12., -12.,  18.])
```

Cache Destruction

To accelerate repeat transforms on arrays of the same shape and dtype, scipy.fftpack keeps a cache of the prime factorization of length of the array and pre-computed trigonometric functions. These caches can be destroyed by calling the appropriate function in `scipy.fftpack._fftpack`. dst(type=1) and idst(type=1) share a cache (`*dst1_cache`). As do dst(type=2), dst(type=3), idst(type=3), and idst(type=3) (`*dst2_cache`).

References

3.1.8 Signal Processing (`scipy.signal`)

The signal processing toolbox currently contains some filtering functions, a limited set of filter design tools, and a few B-spline interpolation algorithms for one- and two-dimensional data. While the B-spline algorithms could technically be placed under the interpolation category, they are included here because they only work with equally-spaced data and make heavy use of filter-theory and transfer-function formalism to provide a fast B-spline transform. To understand this section you will need to understand that a signal in SciPy is an array of real or complex numbers.

B-splines

A B-spline is an approximation of a continuous function over a finite- domain in terms of B-spline coefficients and knot points. If the knot- points are equally spaced with spacing Δx , then the B-spline approximation to a 1-dimensional function is the finite-basis expansion.

$$y\left(x\right)\approx\sum_{j}c_{j}\beta^{o}\left(\frac{x}{\Delta x}-j\right).$$

In two dimensions with knot-spacing Δx and Δy , the function representation is

$$z\left(x,y\right)\approx\sum_{j}\sum_{k}c_{jk}\beta^{o}\left(\frac{x}{\Delta x}-j\right)\beta^{o}\left(\frac{y}{\Delta y}-k\right).$$

In these expressions, $\beta^{o}\left(\cdot\right)$ is the space-limited B-spline basis function of order, o . The requirement of equally-spaced knot-points and equally-spaced data points, allows the development of fast (inverse-filtering) algorithms for

determining the coefficients, c_j , from sample-values, y_n . Unlike the general spline interpolation algorithms, these algorithms can quickly find the spline coefficients for large images.

The advantage of representing a set of samples via B-spline basis functions is that continuous-domain operators (derivatives, re- sampling, integral, etc.) which assume that the data samples are drawn from an underlying continuous function can be computed with relative ease from the spline coefficients. For example, the second-derivative of a spline is

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \beta^{o''}\left(\frac{x}{\Delta x} - j\right).$$

Using the property of B-splines that

$$\frac{d^2 \beta^o(w)}{dw^2} = \beta^{o-2}(w+1) - 2\beta^{o-2}(w) + \beta^{o-2}(w-1)$$

it can be seen that

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \left[\beta^{o-2}\left(\frac{x}{\Delta x} - j + 1\right) - 2\beta^{o-2}\left(\frac{x}{\Delta x} - j\right) + \beta^{o-2}\left(\frac{x}{\Delta x} - j - 1\right)\right].$$

If $o = 3$, then at the sample points,

$$\begin{aligned} \Delta x^2 \left. y'(x) \right|_{x=n\Delta x} &= \sum_j c_j \delta_{n-j+1} - 2c_j \delta_{n-j} + c_j \delta_{n-j-1}, \\ &= c_{n+1} - 2c_n + c_{n-1}. \end{aligned}$$

Thus, the second-derivative signal can be easily calculated from the spline fit. if desired, smoothing splines can be found to make the second-derivative less sensitive to random-errors.

The savvy reader will have already noticed that the data samples are related to the knot coefficients via a convolution operator, so that simple convolution with the sampled B-spline function recovers the original data from the spline coefficients. The output of convolutions can change depending on how boundaries are handled (this becomes increasingly more important as the number of dimensions in the data- set increases). The algorithms relating to B-splines in the signal- processing sub package assume mirror-symmetric boundary conditions. Thus, spline coefficients are computed based on that assumption, and data-samples can be recovered exactly from the spline coefficients by assuming them to be mirror-symmetric also.

Currently the package provides functions for determining second- and third- order cubic spline coefficients from equally spaced samples in one- and two- dimensions (*qspline1d*, *qspline2d*, *cspline1d*, *cspline2d*). The package also supplies a function (*bspline*) for evaluating the bspline basis function, $\beta^o(x)$ for arbitrary order and x. For large o , the B-spline basis function can be approximated well by a zero-mean Gaussian function with standard-deviation equal to $\sigma_o = (o+1)/12$:

$$\beta^o(x) \approx \frac{1}{\sqrt{2\pi\sigma_o^2}} \exp\left(-\frac{x^2}{2\sigma_o}\right).$$

A function to compute this Gaussian for arbitrary x and o is also available (*gauss_spline*). The following code and Figure uses spline-filtering to compute an edge-image (the second-derivative of a smoothed spline) of a raccoon's face which is an array returned by the command `misc.face`. The command *sepfir2d* was used to apply a separable two-dimensional FIR filter with mirror- symmetric boundary conditions to the spline coefficients. This function is ideally suited for reconstructing samples from spline coefficients and is faster than *convolve2d* which convolves arbitrary two-dimensional filters and allows for choosing mirror-symmetric boundary conditions.

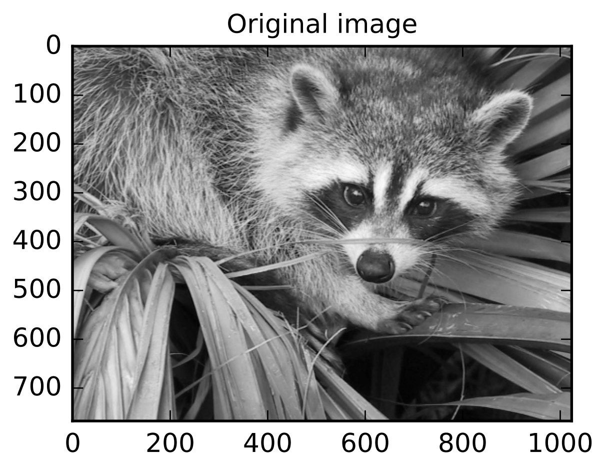
```
>>> import numpy as np
>>> from scipy import signal, misc
>>> import matplotlib.pyplot as plt
```

```
>>> image = misc.face(gray=True).astype(np.float32)
>>> derfilt = np.array([1.0, -2, 1.0], dtype=np.float32)
>>> ck = signal.cspline2d(image, 8.0)
>>> deriv = (signal.sepfir2d(ck, derfilt, [1]) +
...          signal.sepfir2d(ck, [1], derfilt))
```

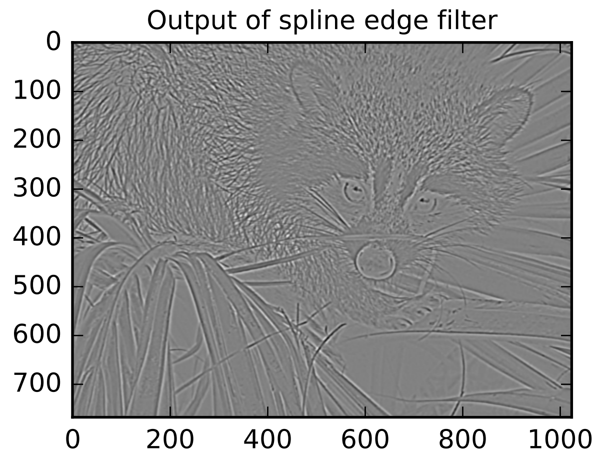
Alternatively we could have done:

```
laplacian = np.array([[0,1,0], [1,-4,1], [0,1,0]], dtype=np.float32)
deriv2 = signal.convolve2d(ck,laplacian,mode='same',boundary='symm')
```

```
>>> plt.figure()
>>> plt.imshow(image)
>>> plt.gray()
>>> plt.title('Original image')
>>> plt.show()
```




```
>>> plt.figure()
>>> plt.imshow(deriv)
>>> plt.gray()
>>> plt.title('Output of spline edge filter')
>>> plt.show()
```



Filtering

Filtering is a generic name for any system that modifies an input signal in some way. In SciPy a signal can be thought of as a Numpy array. There are different kinds of filters for different kinds of operations. There are two broad kinds of filtering operations: linear and non-linear. Linear filters can always be reduced to multiplication of the flattened Numpy array by an appropriate matrix resulting in another flattened Numpy array. Of course, this is not usually the best way to compute the filter as the matrices and vectors involved may be huge. For example filtering a 512×512 image with this method would require multiplication of a $512^2 \times 512^2$ matrix with a 512^2 vector. Just trying to store the $512^2 \times 512^2$ matrix using a standard Numpy array would require $68,719,476,736$ elements. At 4 bytes per element this would require 256GB of memory. In most applications most of the elements of this matrix are zero and a different method for computing the output of the filter is employed.

Convolution/Correlation

Many linear filters also have the property of shift-invariance. This means that the filtering operation is the same at different locations in the signal and it implies that the filtering matrix can be constructed from knowledge of one row (or column) of the matrix alone. In this case, the matrix multiplication can be accomplished using Fourier transforms.

Let $x[n]$ define a one-dimensional signal indexed by the integer n. Full convolution of two one-dimensional signals can be expressed as

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]\,.$$

This equation can only be implemented directly if we limit the sequences to finite support sequences that can be stored in a computer, choose $n = 0$ to be the starting point of both sequences, let $K + 1$ be that value for which $y[n] = 0$ for all $n > K + 1$ and $M + 1$ be that value for which $x[n] = 0$ for all $n > M + 1$, then the discrete convolution expression is

$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k]\, h[n-k]\,.$$

For convenience assume $K \geq M$. Then, more explicitly the output of this operation is

$$\begin{aligned}
y[0] &= x[0]h[0] \\
y[1] &= x[0]h[1]+x[1]h[0] \\
y[2] &= x[0]h[2]+x[1]h[1]+x[2]h[0] \\
\vdots &\quad \vdots \quad \vdots \\
y[M] &= x[0]h[M]+x[1]h[M-1]+\cdots+x[M]h[0] \\
y[M+1] &= x[1]h[M]+x[2]h[M-1]+\cdots+x[M+1]h[0] \\
\vdots &\quad \vdots \quad \vdots \\
y[K] &= x[K-M]h[M]+\cdots+x[K]h[0] \\
y[K+1] &= x[K+1-M]h[M]+\cdots+x[K]h[1] \\
\vdots &\quad \vdots \quad \vdots \\
y[K+M-1] &= x[K-1]h[M]+x[K]h[M-1] \\
y[K+M] &= x[K]h[M].
\end{aligned}$$

Thus, the full discrete convolution of two finite sequences of lengths $K+1$ and $M+1$ respectively results in a finite sequence of length $K+M+1=(K+1)+(M+1)-1$.

One dimensional convolution is implemented in SciPy with the function `convolve`. This function takes as inputs the signals x, h , and two optional flags 'mode' and 'method' and returns the signal y.

The first optional flag 'mode' allows for specification of which part of the output signal to return. The default value of 'full' returns the entire signal. If the flag has a value of 'same' then only the middle K values are returned starting at $y\left[\left\lfloor\frac{M-1}{2}\right\rfloor\right]$ so that the output has the same length as the first input. If the flag has a value of 'valid' then only the middle $K-M+1=(K+1)-(M+1)+1$ output values are returned where z depends on all of the values of the smallest input from $h[0]$ to $h[M]$. In other words only the values $y[M]$ to $y[K]$ inclusive are returned.

The second optional flag 'method' determines how the convolution is computed, either through the Fourier transform approach with `fftconvolve` or through the direct method. By default, it selects the expected faster method. The Fourier transform method has order $O(N\log N)$ while the direct method has order $O(N^2)$. Depending on the big O constant and the value of N, one of these two methods may be faster. The default value 'auto' performs a rough calculation and chooses the expected faster method, while the values 'direct' and 'fft' force computation with the other two methods.

The code below shows a simple example for convolution of 2 sequences

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> h = np.array([0.0, 1.0, 0.0, 0.0, 0.0])
>>> signal.convolve(x, h)
array([ 0.,  1.,  2.,  3.,  0.,  0.,  0.])
>>> signal.convolve(x, h, 'same')
array([ 2.,  3.,  0.])
```

This same function `convolve` can actually take N -dimensional arrays as inputs and will return the N -dimensional convolution of the two arrays as is shown in the code example below. The same input flags are available for that case as well.

```
>>> x = np.array([[1., 1., 0., 0.], [1., 1., 0., 0.], [0., 0., 0., 0.], [0., 0., 0.,␣
↪0.]])
>>> h = np.array([[1., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 1., 0.], [0., 0., 0.,␣
↪0.]])
>>> signal.convolve(x, h)
array([[ 1.,  1.,  0.,  0.,  0.,  0.,  0.],
```

```
       [ 1.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

Correlation is very similar to convolution except for the minus sign becomes a plus sign. Thus

$$w\left[n\right]=\sum_{k=-\infty}^{\infty}y\left[k\right]x\left[n+k\right]$$

is the (cross) correlation of the signals y and x. For finite-length signals with $y\left[n\right]=0$ outside of the range $\left[0,K\right]$ and $x\left[n\right]=0$ outside of the range $\left[0,M\right]$, the summation can simplify to

$$w\left[n\right]=\sum_{k=\max(0,-n)}^{\min(K,M-n)}y\left[k\right]x\left[n+k\right].$$

Assuming again that $K\geq M$ this is

$$\begin{aligned}
w\left[-K\right] & = & y\left[K\right]x\left[0\right]\\
w\left[-K+1\right] & = & y\left[K-1\right]x\left[0\right]+y\left[K\right]x\left[1\right]\\
\vdots & \vdots & \vdots\\
w\left[M-K\right] & = & y\left[K-M\right]x\left[0\right]+y\left[K-M+1\right]x\left[1\right]+\cdots+y\left[K\right]x\left[M\right]\\
w\left[M-K+1\right] & = & y\left[K-M-1\right]x\left[0\right]+\cdots+y\left[K-1\right]x\left[M\right]\\
\vdots & \vdots & \vdots\\
w\left[-1\right] & = & y\left[1\right]x\left[0\right]+y\left[2\right]x\left[1\right]+\cdots+y\left[M+1\right]x\left[M\right]\\
w\left[0\right] & = & y\left[0\right]x\left[0\right]+y\left[1\right]x\left[1\right]+\cdots+y\left[M\right]x\left[M\right]\\
w\left[1\right] & = & y\left[0\right]x\left[1\right]+y\left[1\right]x\left[2\right]+\cdots+y\left[M-1\right]x\left[M\right]\\
w\left[2\right] & = & y\left[0\right]x\left[2\right]+y\left[1\right]x\left[3\right]+\cdots+y\left[M-2\right]x\left[M\right]\\
\vdots & \vdots & \vdots\\
w\left[M-1\right] & = & y\left[0\right]x\left[M-1\right]+y\left[1\right]x\left[M\right]\\
w\left[M\right] & = & y\left[0\right]x\left[M\right].
\end{aligned}$$

The SciPy function `correlate` implements this operation. Equivalent flags are available for this operation to return the full $K+M+1$ length sequence ('full') or a sequence with the same size as the largest sequence starting at $w\left[-K+\left\lfloor\frac{M-1}{2}\right\rfloor\right]$ ('same') or a sequence where the values depend on all the values of the smallest sequence ('valid'). This final option returns the $K-M+1$ values $w\left[M-K\right]$ to $w\left[0\right]$ inclusive.

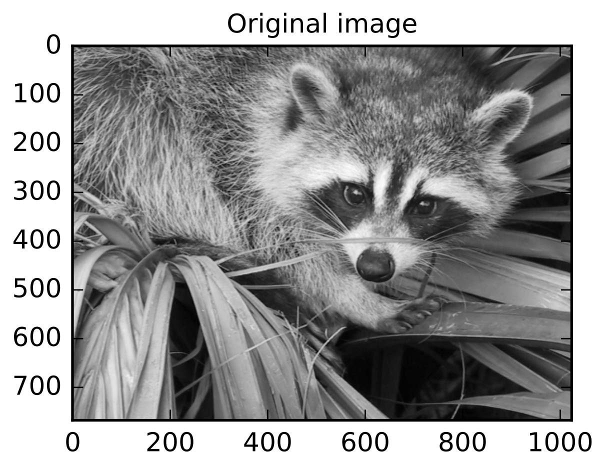
The function `correlate` can also take arbitrary N -dimensional arrays as input and return the N -dimensional convolution of the two arrays on output.

When $N=2$, `correlate` and/or `convolve` can be used to construct arbitrary image filters to perform actions such as blurring, enhancing, and edge-detection for an image.

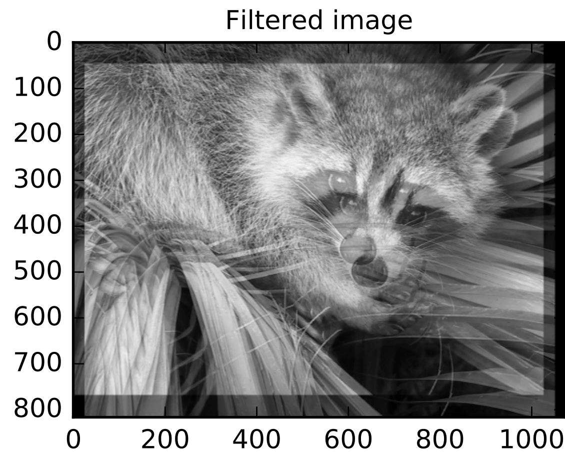
```
>>> import numpy as np
>>> from scipy import signal, misc
>>> import matplotlib.pyplot as plt
```

```
>>> image = misc.face(gray=True)
>>> w = np.zeros((50, 50))
>>> w[0][0] = 1.0
>>> w[49][25] = 1.0
>>> image_new = signal.fftconvolve(image, w)
```

```
>>> plt.figure()
>>> plt.imshow(image)
>>> plt.gray()
>>> plt.title('Original image')
>>> plt.show()
```




```
>>> plt.figure()
>>> plt.imshow(image_new)
>>> plt.gray()
>>> plt.title('Filtered image')
>>> plt.show()
```



Calculating the convolution in the time domain as above is mainly used for filtering when one of the signals is much smaller than the other ($K \gg M$), otherwise linear filtering is more efficiently calculated in the frequency domain provided by the function *fftconvolve*. By default, *convolve* estimates the fastest method using *choose_conv_method*.

If the filter function $w[n, m]$ can be factored according to

$$h[n, m] = h_1[n]h_2[m],$$

convolution can be calculated by means of the function *sepfir2d*. As an example we consider a Gaussian filter *gaussian*

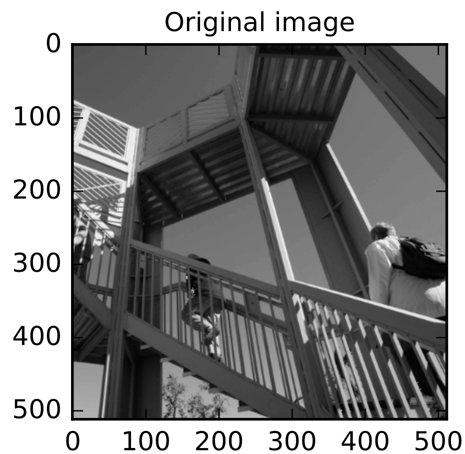
$$h[n, m] \propto e^{-x^2-y^2} = e^{-x^2}e^{-y^2}$$

which is often used for blurring.

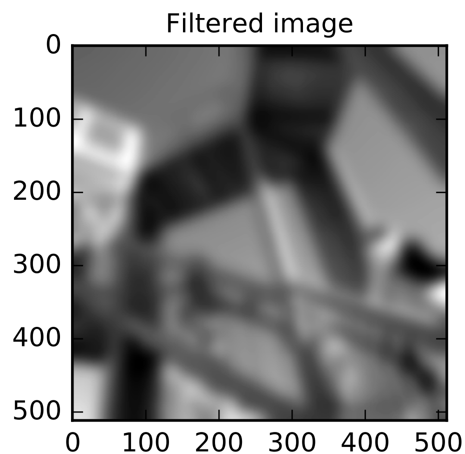
```
>>> import numpy as np
>>> from scipy import signal, misc
>>> import matplotlib.pyplot as plt
```

```
>>> image = misc.ascent()
>>> w = signal.gaussian(50, 10.0)
>>> image_new = signal.sepfir2d(image, w, w)
```

```
>>> plt.figure()
>>> plt.imshow(image)
>>> plt.gray()
>>> plt.title('Original image')
>>> plt.show()
```




```
>>> plt.figure()
>>> plt.imshow(image_new)
>>> plt.gray()
>>> plt.title('Filtered image')
>>> plt.show()
```



Difference-equation filtering

A general class of linear one-dimensional filters (that includes convolution filters) are filters described by the difference equation

$$\sum_{k=0}^{N} a_k y\left[n-k\right] = \sum_{k=0}^{M} b_k x\left[n-k\right]$$

where $x\left[n\right]$ is the input sequence and $y\left[n\right]$ is the output sequence. If we assume initial rest so that $y\left[n\right] = 0$ for $n < 0$, then this kind of filter can be implemented using convolution. However, the convolution filter sequence $h\left[n\right]$ could

be infinite if $a_k \neq 0$ for $k \geq 1$. In addition, this general class of linear filter allows initial conditions to be placed on $y[n]$ for $n < 0$ resulting in a filter that cannot be expressed using convolution.

The difference equation filter can be thought of as finding $y[n]$ recursively in terms of it's previous values

$$a_0 y[n] = -a_1 y[n-1] - \cdots - a_N y[n-N] + \cdots + b_0 x[n] + \cdots + b_M x[n-M].$$

Often $a_0 = 1$ is chosen for normalization. The implementation in SciPy of this general difference equation filter is a little more complicated then would be implied by the previous equation. It is implemented so that only one signal needs to be delayed. The actual implementation equations are (assuming $a_0 = 1$).

$$\begin{aligned}
y[n] &= b_0 x[n] + z_0[n-1] \\
z_0[n] &= b_1 x[n] + z_1[n-1] - a_1 y[n] \\
z_1[n] &= b_2 x[n] + z_2[n-1] - a_2 y[n] \\
\vdots &\quad \vdots \quad \vdots \\
z_{K-2}[n] &= b_{K-1} x[n] + z_{K-1}[n-1] - a_{K-1} y[n] \\
z_{K-1}[n] &= b_K x[n] - a_K y[n],
\end{aligned}$$

where $K = \max(N, M)$. Note that $b_K = 0$ if $K > M$ and $a_K = 0$ if $K > N$. In this way, the output at time n depends only on the input at time n and the value of z_0 at the previous time. This can always be calculated as long as the K values $z_0[n-1] \ldots z_{K-1}[n-1]$ are computed and stored at each time step.

The difference-equation filter is called using the command `lfilter` in SciPy. This command takes as inputs the vector b, the vector, a, a signal x and returns the vector y (the same length as x) computed using the equation given above. If x is N -dimensional, then the filter is computed along the axis provided. If, desired, initial conditions providing the values of $z_0[-1]$ to $z_{K-1}[-1]$ can be provided or else it will be assumed that they are all zero. If initial conditions are provided, then the final conditions on the intermediate variables are also returned. These could be used, for example, to restart the calculation in the same state.

Sometimes it is more convenient to express the initial conditions in terms of the signals $x[n]$ and $y[n]$. In other words, perhaps you have the values of $x[-M]$ to $x[-1]$ and the values of $y[-N]$ to $y[-1]$ and would like to determine what values of $z_m[-1]$ should be delivered as initial conditions to the difference-equation filter. It is not difficult to show that for $0 \leq m < K$,

$$z_m[n] = \sum_{p=0}^{K-m-1} \left(b_{m+p+1} x[n-p] - a_{m+p+1} y[n-p]\right).$$

Using this formula we can find the initial condition vector $z_0[-1]$ to $z_{K-1}[-1]$ given initial conditions on y (and x). The command `lfiltic` performs this function.

As an example consider the following system:

$$y[n] = \frac{1}{2}x[n] + \frac{1}{4}x[n-1] + \frac{1}{3}y[n-1]$$

The code calculates the signal $y[n]$ for a given signal $x[n]$; first for initial conditions $y[-1] = 0$ (default case), then for $y[-1] = 2$ by means of `lfiltic`.

```
>>> import numpy as np
>>> from scipy import signal
```

```
>>> x = np.array([1., 0., 0., 0.])
>>> b = np.array([1.0/2, 1.0/4])
>>> a = np.array([1.0, -1.0/3])
>>> signal.lfilter(b, a, x)
array([0.5, 0.41666667, 0.13888889, 0.0462963])
```

```
>>> zi = signal.lfiltic(b, a, y=[2.])
>>> signal.lfilter(b, a, x, zi=zi)
(array([ 1.16666667,  0.63888889,  0.21296296,  0.07098765]), array([0.02366]))
```

Note that the output signal $y[n]$ has the same length as the length as the input signal $x[n]$.

Analysis of Linear Systems

Linear system described a linear difference equation can be fully described by the coefficient vectors a and b as was done above; an alternative representation is to provide a factor k, N_z zeros z_k and N_p poles p_k, respectively, to describe the system by means of its transfer function $H(z)$ according to

$$H(z) = k\frac{(z - z_1)(z - z_2)...(z - z_{N_z})}{(z - p_1)(z - p_2)...(z - p_{N_p})}$$

This alternative representation can be obtain with the scipy function *tf2zpk*; the inverse is provided by *zpk2tf*.

For the example from above we have

```
>>> b = np.array([1.0/2, 1.0/4])
>>> a = np.array([1.0, -1.0/3])
>>> signal.tf2zpk(b, a)
(array([-0.5]), array([ 0.33333333]), 0.5)
```

i.e. the system has a zero at $z = -1/2$ and a pole at $z = 1/3$.

The scipy function *freqz* allows calculation of the frequency response of a system described by the coefficients a_k and b_k. See the help of the *freqz* function of a comprehensive example.

Filter Design

Time-discrete filters can be classified into finite response (FIR) filters and infinite response (IIR) filters. FIR filters can provide a linear phase response, whereas IIR filters cannot. Scipy provides functions for designing both types of filters.

FIR Filter

The function *firwin* designs filters according to the window method. Depending on the provided arguments, the function returns different filter types (e.g. low-pass, band-pass...).

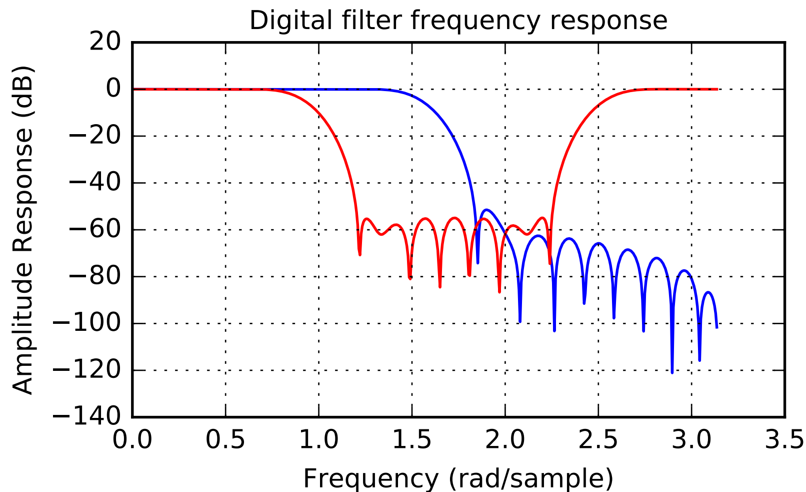
The example below designs a low-pass and a band-stop filter, respectively.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b1 = signal.firwin(40, 0.5)
>>> b2 = signal.firwin(41, [0.3, 0.8])
>>> w1, h1 = signal.freqz(b1)
>>> w2, h2 = signal.freqz(b2)
```

```
>>> plt.title('Digital filter frequency response')
>>> plt.plot(w1, 20*np.log10(np.abs(h1)), 'b')
>>> plt.plot(w2, 20*np.log10(np.abs(h2)), 'r')
>>> plt.ylabel('Amplitude Response (dB)')
```

```
>>> plt.xlabel('Frequency (rad/sample)')
>>> plt.grid()
>>> plt.show()
```



Note that *firwin* uses per default a normalized frequency defined such that the value 1 corresponds to the Nyquist frequency, whereas the function *freqz* is defined such that the value π corresponds to the Nyquist frequency.

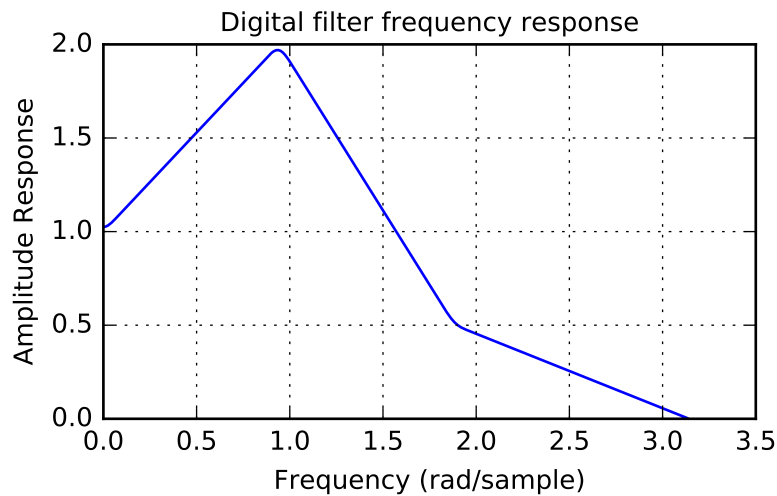
The function *firwin2* allows design of almost arbitrary frequency responses by specifying an array of corner frequencies and corresponding gains, respectively.

The example below designs a filter with such an arbitrary amplitude response.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b = signal.firwin2(150, [0.0, 0.3, 0.6, 1.0], [1.0, 2.0, 0.5, 0.0])
>>> w, h = signal.freqz(b)
```

```
>>> plt.title('Digital filter frequency response')
>>> plt.plot(w, np.abs(h))
>>> plt.title('Digital filter frequency response')
>>> plt.ylabel('Amplitude Response')
>>> plt.xlabel('Frequency (rad/sample)')
>>> plt.grid()
>>> plt.show()
```



Note the linear scaling of the y-axis and the different definition of the Nyquist frequency in *firwin2* and *freqz* (as explained above).

IIR Filter

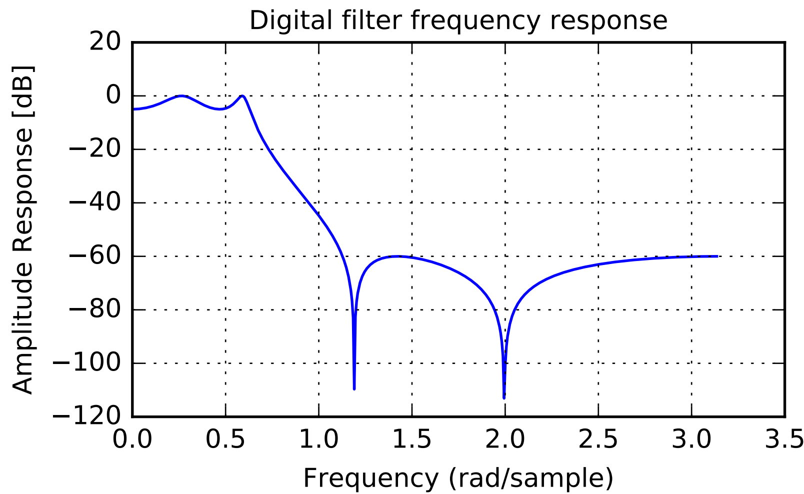
Scipy provides two functions to directly design IIR *iirdesign* and *iirfilter* where the filter type (e.g. elliptic) is passed as an argument and several more filter design functions for specific filter types; e.g. *ellip*.

The example below designs an elliptic low-pass filter with defined passband and stopband ripple, respectively. Note the much lower filter order (order 4) compared with the FIR filters from the examples above in order to reach the same stop-band attenuation of ≈ 60 dB.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.iirfilter(4, Wn=0.2, rp=5, rs=60, btype='lowpass', ftype='ellip')
>>> w, h = signal.freqz(b, a)
```

```
>>> plt.title('Digital filter frequency response')
>>> plt.plot(w, 20*np.log10(np.abs(h)))
>>> plt.title('Digital filter frequency response')
>>> plt.ylabel('Amplitude Response [dB]')
>>> plt.xlabel('Frequency (rad/sample)')
>>> plt.grid()
>>> plt.show()
```



Filter Coefficients

Filter coefficients can be stored in several different formats:

- 'ba' or 'tf' = transfer function coefficients
- 'zpk' = zeros, poles, and overall gain
- 'ss' = state-space system representation
- 'sos' = transfer function coefficients of second-order sections

Functions such as `tf2zpk` and `zpk2ss` can convert between them.

Transfer function representation

The `ba` or `tf` format is a 2-tuple `(b, a)` representing a transfer function, where *b* is a length `M+1` array of coefficients of the *M*-order numerator polynomial, and *a* is a length `N+1` array of coefficients of the *N*-order denominator, as positive, descending powers of the transfer function variable. So the tuple of $b = [b_0, b_1, ..., b_M]$ and $a = [a_0, a_1, ..., a_N]$ can represent an analog filter of the form:

$$H(s) = \frac{b_0 s^M + b_1 s^{(M-1)} + \cdots + b_M}{a_0 s^N + a_1 s^{(N-1)} + \cdots + a_N} = \frac{\sum_{i=0}^{M} b_i s^{(M-i)}}{\sum_{i=0}^{N} a_i s^{(N-i)}}$$

or a discrete-time filter of the form:

$$H(z) = \frac{b_0 z^M + b_1 z^{(M-1)} + \cdots + b_M}{a_0 z^N + a_1 z^{(N-1)} + \cdots + a_N} = \frac{\sum_{i=0}^{M} b_i z^{(M-i)}}{\sum_{i=0}^{N} a_i z^{(N-i)}}$$

This "positive powers" form is found more commonly in controls engineering. If *M* and *N* are equal (which is true for all filters generated by the bilinear transform), then this happens to be equivalent to the "negative powers" discrete-time form preferred in DSP:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}} = \frac{\sum_{i=0}^{M} b_i z^{-i}}{\sum_{i=0}^{N} a_i z^{-i}}$$

Although this is true for common filters, remember that this is not true in the general case. If *M* and *N* are not equal, the discrete-time transfer function coefficients must first be converted to the “positive powers” form before finding the poles and zeros.

This representation suffers from numerical error at higher orders, so other formats are preferred when possible.

Zeros and poles representation

The `zpk` format is a 3-tuple `(z, p, k)`, where z is an *M*-length array of the complex zeros of the transfer function $z = [z_0, z_1, ..., z_{M-1}]$, *p* is an *N*-length array of the complex poles of the transfer function $p = [p_0, p_1, ..., p_{N-1}]$, and *k* is a scalar gain. These represent the digital transfer function:

$$H(z) = k \cdot \frac{(z - z_0)(z - z_1)\cdots(z - z_{(M-1)})}{(z - p_0)(z - p_1)\cdots(z - p_{(N-1)})} = k\frac{\prod_{i=0}^{M-1}(z - z_i)}{\prod_{i=0}^{N-1}(z - p_i)}$$

or the analog transfer function:

$$H(s) = k \cdot \frac{(s - z_0)(s - z_1)\cdots(s - z_{(M-1)})}{(s - p_0)(s - p_1)\cdots(s - p_{(N-1)})} = k\frac{\prod_{i=0}^{M-1}(s - z_i)}{\prod_{i=0}^{N-1}(s - p_i)}$$

Although the sets of roots are stored as ordered NumPy arrays, their ordering does not matter; `([-1, -2], [-3, -4], 1)` is the same filter as `([-2, -1], [-4, -3], 1)`.

State-space system representation

The `ss` format is a 4-tuple of arrays `(A, B, C, D)` representing the state-space of an *N*-order digital/discrete-time system of the form:

$$\begin{aligned} \mathbf{x}[k+1] &= A\mathbf{x}[k] + B\mathbf{u}[k] \\ \mathbf{y}[k] &= C\mathbf{x}[k] + D\mathbf{u}[k] \end{aligned}$$

or a continuous/analog system of the form:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) + D\mathbf{u}(t) \end{aligned}$$

with *P* inputs, *Q* outputs and *N* state variables, where:

- *x* is the state vector
- *y* is the output vector of length *Q*
- *u* is the input vector of length *P*
- *A* is the state matrix, with shape `(N, N)`
- *B* is the input matrix with shape `(N, P)`
- *C* is the output matrix with shape `(Q, N)`
- *D* is the feedthrough or feedforward matrix with shape `(Q, P)`. (In cases where the system does not have a direct feedthrough, all values in *D* are zero.)

State-space is the most general representation, and the only one that allows for multiple-input, multiple-output (MIMO) systems. There are multiple state-space representations for a given transfer function. Specifically, the “controllable canonical form” and “observable canonical form” have the same coefficients as the `tf` representation, and therefore suffer from the same numerical errors.

Second-order sections representation

The `sos` format is a single 2D array of shape `(n_sections, 6)`, representing a sequence of second-order transfer functions which, when cascaded in series, realize a higher-order filter with minimal numerical error. Each row corresponds to a second-order `tf` representation, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients:

$$[b_0, b_1, b_2, a_0, a_1, a_2]$$

The coefficients are typically normalized such that a_0 is always 1. The section order is usually not important with floating-point computation; the filter output will be the same regardless.

Filter transformations

The IIR filter design functions first generate a prototype analog lowpass filter with a normalized cutoff frequency of 1 rad/sec. This is then transformed into other frequencies and band types using the following substitutions:

Type	Transformation
lp2lp	$s \rightarrow \frac{s}{\omega_0}$
lp2hp	$s \rightarrow \frac{\omega_0}{s}$
lp2bp	$s \rightarrow \frac{s^2+\omega_0{}^2}{s \cdot \mathrm{BW}}$
lp2bs	$s \rightarrow \frac{s \cdot \mathrm{BW}}{s^2+\omega_0{}^2}$

Here, ω_0 is the new cutoff or center frequency, and BW is the bandwidth. These preserve symmetry on a logarithmic frequency axis.

To convert the transformed analog filter into a digital filter, the *bilinear* transform is used, which makes the following substitution:

$$s \rightarrow \frac{2}{T} \frac{z-1}{z+1}$$

where T is the sampling time (the inverse of the sampling frequency).

Other filters

The signal processing package provides many more filters as well.

Median Filter

A median filter is commonly applied when noise is markedly non-Gaussian or when it is desired to preserve edges. The median filter works by sorting all of the array pixel values in a rectangular region surrounding the point of interest. The sample median of this list of neighborhood pixel values is used as the value for the output array. The sample median is the middle array value in a sorted list of neighborhood values. If there are an even number of elements in the neighborhood, then the average of the middle two values is used as the median. A general purpose median filter that works on N-dimensional arrays is *medfilt* . A specialized version that works only for two-dimensional arrays is available as *medfilt2d* .

Order Filter

A median filter is a specific example of a more general class of filters called order filters. To compute the output at a particular pixel, all order filters use the array values in a region surrounding that pixel. These array values are sorted and then one of them is selected as the output value. For the median filter, the sample median of the list of array values

is used as the output. A general order filter allows the user to select which of the sorted values will be used as the output. So, for example one could choose to pick the maximum in the list or the minimum. The order filter takes an additional argument besides the input array and the region mask that specifies which of the elements in the sorted list of neighbor array values should be used as the output. The command to perform an order filter is `order_filter`.

Wiener filter

The Wiener filter is a simple deblurring filter for denoising images. This is not the Wiener filter commonly described in image reconstruction problems but instead it is a simple, local-mean filter. Let x be the input signal, then the output is

$$y = \left\{ \begin{array}{cc} \frac{\sigma^2}{\sigma_x^2} m_x + \left(1 - \frac{\sigma^2}{\sigma_x^2}\right) x & \sigma_x^2 \geq \sigma^2, \\ m_x & \sigma_x^2 < \sigma^2, \end{array} \right.$$

where m_x is the local estimate of the mean and σ_x^2 is the local estimate of the variance. The window for these estimates is an optional input parameter (default is 3×3). The parameter σ^2 is a threshold noise parameter. If σ is not given then it is estimated as the average of the local variances.

Hilbert filter

The Hilbert transform constructs the complex-valued analytic signal from a real signal. For example if $x = \cos \omega n$ then $y = \text{hilbert}(x)$ would return (except near the edges) $y = \exp(j\omega n)$. In the frequency domain, the hilbert transform performs

$$Y = X \cdot H$$

where H is 2 for positive frequencies, 0 for negative frequencies and 1 for zero-frequencies.

Analog Filter Design

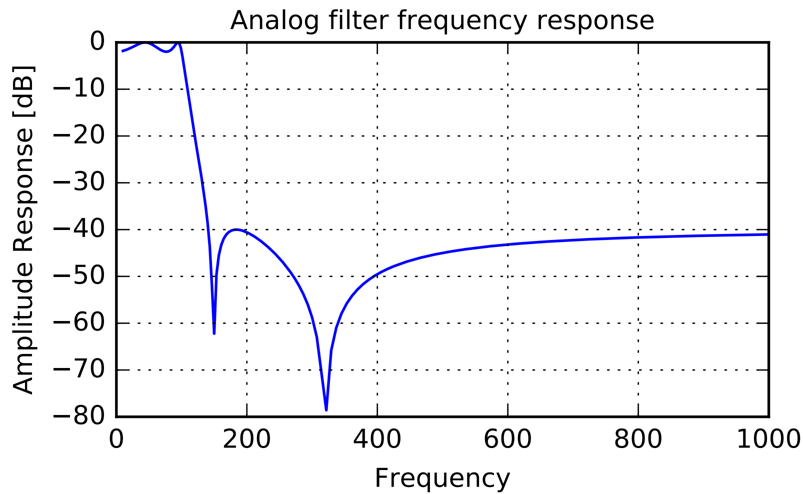
The functions `iirdesign`, `iirfilter`, and the filter design functions for specific filter types (e.g. `ellip`) all have a flag *analog* which allows design of analog filters as well.

The example below designs an analog (IIR) filter, obtains via `tf2zpk` the poles and zeros and plots them in the complex s-plane. The zeros at $\omega \approx 150$ and $\omega \approx 300$ can be clearly seen in the amplitude response.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.iirdesign(wp=100, ws=200, gpass=2.0, gstop=40., analog=True)
>>> w, h = signal.freqs(b, a)
```

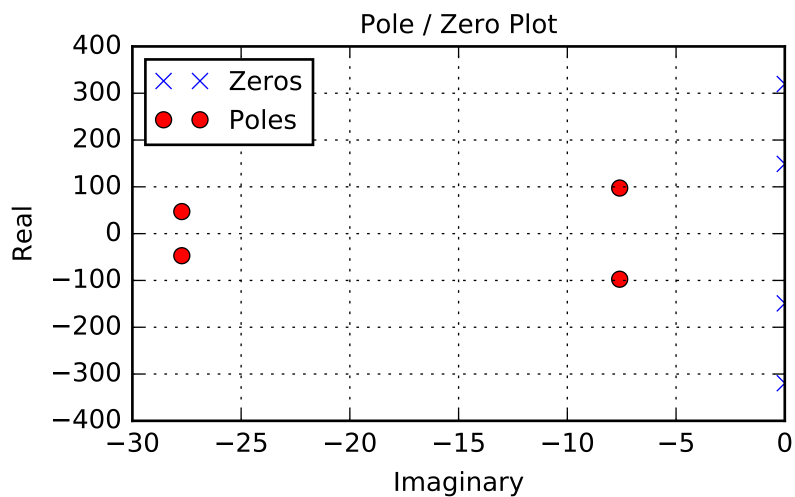
```
>>> plt.title('Analog filter frequency response')
>>> plt.plot(w, 20*np.log10(np.abs(h)))
>>> plt.ylabel('Amplitude Response [dB]')
>>> plt.xlabel('Frequency')
>>> plt.grid()
>>> plt.show()
```




```
>>> z, p, k = signal.tf2zpk(b, a)
```

```
>>> plt.plot(np.real(z), np.imag(z), 'xb')
>>> plt.plot(np.real(p), np.imag(p), 'or')
>>> plt.legend(['Zeros', 'Poles'], loc=2)
```

```
>>> plt.title('Pole / Zero Plot')
>>> plt.ylabel('Real')
>>> plt.xlabel('Imaginary')
>>> plt.grid()
>>> plt.show()
```



Spectral Analysis

Periodogram Measurements

The scipy function `periodogram` provides a method to estimate the spectral density using the periodogram method.

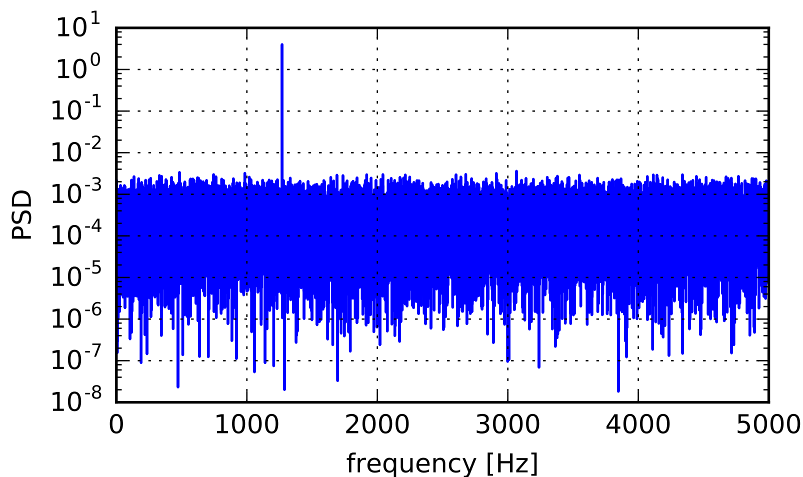
The example below calculates the periodogram of a sine signal in white Gaussian noise.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1270.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
```

```
>>> f, Pper_spec = signal.periodogram(x, fs, 'flattop', scaling='spectrum')
```

```
>>> plt.semilogy(f, Pper_spec)
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD')
>>> plt.grid()
>>> plt.show()
```



Spectral Analysis using Welch's Method

An improved method, especially with respect to noise immunity, is Welch's method which is implemented by the scipy function `welch`.

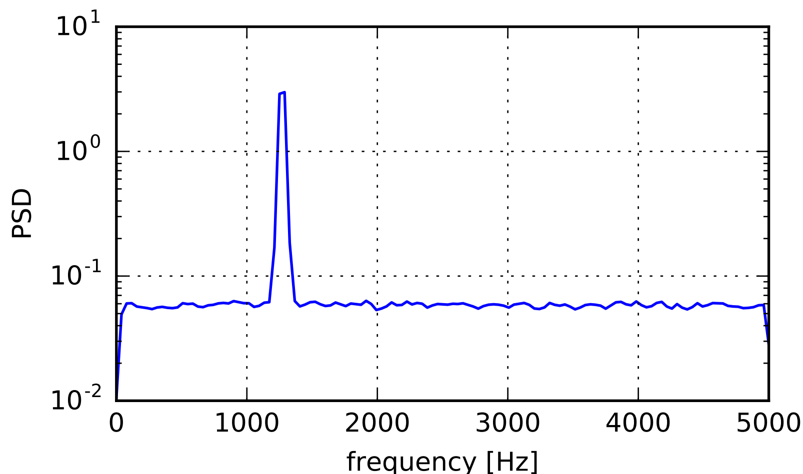
The example below estimates the spectrum using Welch's method and uses the same parameters as the example above. Note the much smoother noise floor of the spectrogram.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1270.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
```

```
>>> f, Pwelch_spec = signal.welch(x, fs, scaling='spectrum')
```

```
>>> plt.semilogy(f, Pwelch_spec)
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD')
>>> plt.grid()
>>> plt.show()
```



Lomb-Scargle Periodograms (`lombscargle`)

Least-squares spectral analysis (LSSA) is a method of estimating a frequency spectrum, based on a least squares fit of sinusoids to data samples, similar to Fourier analysis. Fourier analysis, the most used spectral method in science, generally boosts long-periodic noise in long gapped records; LSSA mitigates such problems.

The Lomb-Scargle method performs spectral analysis on unevenly sampled data and is known to be a powerful way to find, and test the significance of, weak periodic signals.

For a time series comprising N_t measurements $X_j \equiv X(t_j)$ sampled at times t_j where $(j = 1, \dots, N_t)$, assumed to have been scaled and shifted such that its mean is zero and its variance is unity, the normalized Lomb-Scargle periodogram at frequency f is

$$P_n(f)\frac{1}{2}\left\{\frac{\left[\sum_j^{N_t} X_j \cos\omega(t_j-\tau)\right]^2}{\sum_j^{N_t}\cos^2\omega(t_j-\tau)}+\frac{\left[\sum_j^{N_t} X_j \sin\omega(t_j-\tau)\right]^2}{\sum_j^{N_t}\sin^2\omega(t_j-\tau)}\right\}.$$

Here, $\omega \equiv 2\pi f$ is the angular frequency. The frequency dependent time offset τ is given by

$$\tan 2\omega\tau = \frac{\sum_j^{N_t} \sin 2\omega t_j}{\sum_j^{N_t} \cos 2\omega t_j}.$$

The *lombscargle* function calculates the periodogram using a slightly modified algorithm due to Townsend[3] which allows the periodogram to be calculated using only a single pass through the input arrays for each frequency.

The equation is refactored as:

$$P_n(f) = \frac{1}{2}\left[\frac{(c_\tau XC + s_\tau XS)^2}{c_\tau^2 CC + 2c_\tau s_\tau CS + s_\tau^2 SS} + \frac{(c_\tau XS - s_\tau XC)^2}{c_\tau^2 SS - 2c_\tau s_\tau CS + s_\tau^2 CC}\right]$$

and

$$\tan 2\omega\tau = \frac{2CS}{CC - SS}.$$

Here,

$$c_\tau = \cos\omega\tau, \qquad s_\tau = \sin\omega\tau$$

while the sums are

$$\begin{aligned}
XC &= \sum_j^{N_t} X_j \cos\omega t_j \\
XS &= \sum_j^{N_t} X_j \sin\omega t_j \\
CC &= \sum_j^{N_t} \cos^2\omega t_j \\
SS &= \sum_j^{N_t} \sin^2\omega t_j \\
CS &= \sum_j^{N_t} \cos\omega t_j \sin\omega t_j.
\end{aligned}$$

This requires $N_f(2N_t + 3)$ trigonometric function evaluations giving a factor of ~ 2 speed increase over the straightforward implementation.

Detrend

Scipy provides the function *detrend* to remove a constant or linear trend in a data series in order to see effect of higher order.

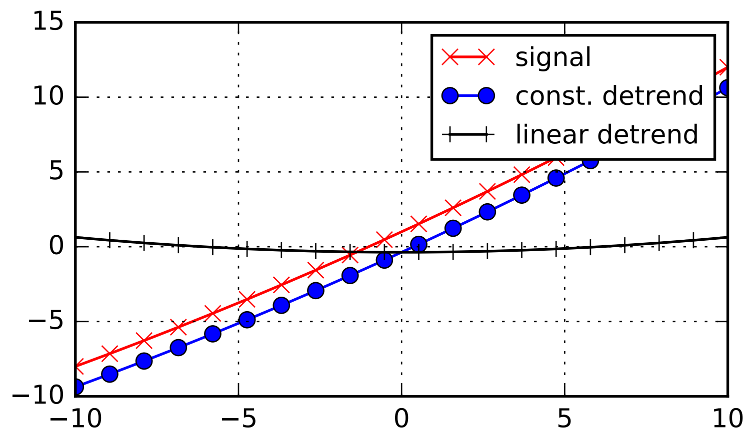
The example below removes the constant and linear trend of a 2-nd order polynomial time series and plots the remaining signal components.

```
>>> import numpy as np
>>> import scipy.signal as signal
>>> import matplotlib.pyplot as plt
```

[3] R.H.D. Townsend, "Fast calculation of the Lomb-Scargle periodogram using graphics processing units.", The Astrophysical Journal Supplement Series, vol 191, pp. 247-253, 2010

```
>>> t = np.linspace(-10, 10, 20)
>>> y = 1 + t + 0.01*t**2
>>> yconst = signal.detrend(y, type='constant')
>>> ylin = signal.detrend(y, type='linear')
```

```
>>> plt.plot(t, y, '-rx')
>>> plt.plot(t, yconst, '-bo')
>>> plt.plot(t, ylin, '-k+')
>>> plt.grid()
>>> plt.legend(['signal', 'const. detrend', 'linear detrend'])
>>> plt.show()
```



References

Some further reading and related software:

3.1.9 Linear Algebra (`scipy.linalg`)

When SciPy is built using the optimized ATLAS LAPACK and BLAS libraries, it has very fast linear algebra capabilities. If you dig deep enough, all of the raw lapack and blas libraries are available for your use for even more speed. In this section, some easier-to-use interfaces to these routines are described.

All of these linear algebra routines expect an object that can be converted into a 2-dimensional array. The output of these routines is also a two-dimensional array.

`scipy.linalg` contains all the functions in `numpy.linalg`. plus some other more advanced ones not contained in `numpy.linalg`

Another advantage of using `scipy.linalg` over `numpy.linalg` is that it is always compiled with BLAS/LAPACK support, while for numpy this is optional. Therefore, the scipy version might be faster depending on how numpy was installed.

Therefore, unless you don't want to add `scipy` as a dependency to your `numpy` program, use `scipy.linalg` instead of `numpy.linalg`

numpy.matrix vs 2D numpy.ndarray

The classes that represent matrices, and basic operations such as matrix multiplications and transpose are a part of `numpy`. For convenience, we summarize the differences between `numpy.matrix` and `numpy.ndarray` here.

`numpy.matrix` is matrix class that has a more convenient interface than `numpy.ndarray` for matrix operations. This class supports for example MATLAB-like creation syntax via the, has matrix multiplication as default for the `*` operator, and contains `I` and `T` members that serve as shortcuts for inverse and transpose:

```
>>> import numpy as np
>>> A = np.mat('[1 2;3 4]')
>>> A
matrix([[1, 2],
        [3, 4]])
>>> A.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> b = np.mat('[5 6]')
>>> b
matrix([[5, 6]])
>>> b.T
matrix([[5],
        [6]])
>>> A*b.T
matrix([[17],
        [39]])
```

Despite its convenience, the use of the `numpy.matrix` class is discouraged, since it adds nothing that cannot be accomplished with 2D `numpy.ndarray` objects, and may lead to a confusion of which class is being used. For example, the above code can be rewritten as:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
      [3, 4]])
>>> linalg.inv(A)
array([[-2. ,  1. ],
      [ 1.5, -0.5]])
>>> b = np.array([[5,6]]) #2D array
>>> b
array([[5, 6]])
>>> b.T
array([[5],
      [6]])
>>> A*b #not matrix multiplication!
array([[ 5, 12],
      [15, 24]])
>>> A.dot(b.T) #matrix multiplication
array([[17],
      [39]])
>>> b = np.array([5,6]) #1D array
>>> b
array([5, 6])
>>> b.T  #not matrix transpose!
array([5, 6])
>>> A.dot(b)  #does not matter for multiplication
```

```
array([17, 39])
```

`scipy.linalg` operations can be applied equally to `numpy.matrix` or to 2D `numpy.ndarray` objects.

Basic routines

Finding Inverse

The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ such that $\mathbf{AB} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix consisting of ones down the main diagonal. Usually $\mathbf{B}$ is denoted $\mathbf{B} = \mathbf{A}^{-1}$. In SciPy, the matrix inverse of the Numpy array, A, is obtained using `linalg.inv (A)` , or using `A.I` if `A` is a Matrix. For example, let

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

then

$$\mathbf{A^{-1}} = \frac{1}{25}\begin{bmatrix} -37 & 9 & 22 \\ 14 & 2 & -9 \\ 4 & -3 & 1 \end{bmatrix} = \begin{bmatrix} -1.48 & 0.36 & 0.88 \\ 0.56 & 0.08 & -0.36 \\ 0.16 & -0.12 & 0.04 \end{bmatrix}.$$

The following example demonstrates this computation in SciPy

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,3,5],[2,5,1],[2,3,8]])
>>> A
array([[1, 3, 5],
       [2, 5, 1],
       [2, 3, 8]])
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
>>> A.dot(linalg.inv(A)) #double check
array([[  1.00000000e+00,  -1.11022302e-16,  -5.55111512e-17],
       [  3.05311332e-16,   1.00000000e+00,   1.87350135e-16],
       [  2.22044605e-16,  -1.11022302e-16,   1.00000000e+00]])
```

Solving linear system

Solving linear systems of equations is straightforward using the scipy command `linalg.solve`. This command expects an input matrix and a right-hand-side vector. The solution vector is then computed. An option for entering a symmetric matrix is offered which can speed up the processing when applicable. As an example, suppose it is desired to solve the following simultaneous equations:

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned}$$

We could find the solution vector using a matrix inverse:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25}\begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}.$$

However, it is better to use the linalg.solve command which can be faster and more numerically stable. In this case it however gives the same answer as shown in the following example:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5], [6]])
>>> b
array([[5],
       [6]])
>>> linalg.inv(A).dot(b)   # slow
array([[-4. ],
       [ 4.5]])
>>> A.dot(linalg.inv(A).dot(b)) - b   # check
array([[  8.88178420e-16],
       [  2.66453526e-15]])
>>> np.linalg.solve(A, b)   # fast
array([[-4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A, b)) - b   # check
array([[ 0.],
       [ 0.]])
```

Finding Determinant

The determinant of a square matrix $\mathbf{A}$ is often denoted $|\mathbf{A}|$ and is a quantity often used in linear algebra. Suppose a_{ij} are the elements of the matrix $\mathbf{A}$ and let $M_{ij} = |\mathbf{A}_{ij}|$ be the determinant of the matrix left by removing the ith row and j^{th} column from $\mathbf{A}$. Then for any row i,

$$|\mathbf{A}| = \sum_j (-1)^{i+j} a_{ij} M_{ij}.$$

This is a recursive way to define the determinant where the base case is defined by accepting that the determinant of a 1×1 matrix is the only matrix element. In SciPy the determinant can be calculated with `linalg.det` . For example, the determinant of

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

is

$$\begin{aligned} |\mathbf{A}| &= 1 \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1 (5 \cdot 8 - 3 \cdot 1) - 3 (2 \cdot 8 - 2 \cdot 1) + 5 (2 \cdot 3 - 2 \cdot 5) = -25. \end{aligned}$$

In SciPy this is computed as shown in this example:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.det(A)
-2.0
```

Computing norms

Matrix and vector norms can also be computed with SciPy. A wide range of norm definitions are available using different parameters to the order argument of `linalg.norm` . This function takes a rank-1 (vectors) or a rank-2 (matrices) array and an optional order argument (default is 2). Based on these inputs a vector or matrix norm of the requested order is computed.

For vector x , the order parameter can be any real number including `inf` or `-inf`. The computed norm is

$$\|\mathbf{x}\| = \begin{cases} \max|x_i| & \text{ord} = \text{inf} \\ \min|x_i| & \text{ord} = -\text{inf} \\ \left(\sum_i |x_i|^{\text{ord}}\right)^{1/\text{ord}} & |\text{ord}| < \infty. \end{cases}$$

For matrix $\mathbf{A}$ the only valid values for norm are $\pm 2, \pm 1, \pm$ inf, and 'fro' (or 'f') Thus,

$$\|\mathbf{A}\| = \begin{cases} \max_i \sum_j |a_{ij}| & \text{ord} = \text{inf} \\ \min_i \sum_j |a_{ij}| & \text{ord} = -\text{inf} \\ \max_j \sum_i |a_{ij}| & \text{ord} = 1 \\ \min_j \sum_i |a_{ij}| & \text{ord} = -1 \\ \max \sigma_i & \text{ord} = 2 \\ \min \sigma_i & \text{ord} = -2 \\ \sqrt{\text{trace}\left(\mathbf{A}^H\mathbf{A}\right)} & \text{ord} = \text{'fro'} \end{cases}$$

where σ_i are the singular values of $\mathbf{A}$.

Examples:

```
>>> import numpy as np
>>> from scipy import linalg
>>> A=np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> linalg.norm(A)
5.4772255750516612
>>> linalg.norm(A,'fro') # frobenius norm is the default
5.4772255750516612
>>> linalg.norm(A,1) # L1 norm (max column sum)
6
>>> linalg.norm(A,-1)
4
>>> linalg.norm(A,np.inf) # L inf norm (max row sum)
7
```

Solving linear least-squares problems and pseudo-inverses

Linear least-squares problems occur in many branches of applied mathematics. In this problem a set of linear scaling coefficients is sought that allow a model to fit data. In particular it is assumed that data y_i is related to data $\mathbf{x}_i$ through a set of coefficients c_j and model functions $f_j(\mathbf{x}_i)$ via the model

$$y_i = \sum_j c_j f_j(\mathbf{x}_i) + \epsilon_i$$

where ϵ_i represents uncertainty in the data. The strategy of least squares is to pick the coefficients c_j to minimize

$$J(\mathbf{c}) = \sum_i \left| y_i - \sum_j c_j f_j(x_i) \right|^2.$$

Theoretically, a global minimum will occur when

$$\frac{\partial J}{\partial c_n^*} = 0 = \sum_i \left(y_i - \sum_j c_j f_j(x_i) \right) \left(-f_n^*(x_i) \right)$$

or

$$\begin{aligned} \sum_j c_j \sum_i f_j(x_i) f_n^*(x_i) &= \sum_i y_i f_n^*(x_i) \\ \mathbf{A}^H \mathbf{A} \mathbf{c} &= \mathbf{A}^H \mathbf{y} \end{aligned}$$

where

$$\{\mathbf{A}\}_{ij} = f_j(x_i).$$

When $\mathbf{A^H A}$ is invertible, then

$$\mathbf{c} = \left(\mathbf{A}^H \mathbf{A} \right)^{-1} \mathbf{A}^H \mathbf{y} = \mathbf{A}^\dagger \mathbf{y}$$

where $\mathbf{A}^\dagger$ is called the pseudo-inverse of $\mathbf{A}$. Notice that using this definition of $\mathbf{A}$ the model can be written

$$\mathbf{y} = \mathbf{A}\mathbf{c} + \boldsymbol{\epsilon}.$$

The command `linalg.lstsq` will solve the linear least squares problem for $\mathbf{c}$ given $\mathbf{A}$ and $\mathbf{y}$. In addition `linalg.pinv` or `linalg.pinv2` (uses a different method based on singular value decomposition) will find $\mathbf{A}^\dagger$ given $\mathbf{A}$.

The following example and figure demonstrate the use of `linalg.lstsq` and `linalg.pinv` for solving a data-fitting problem. The data shown below were generated using the model:

$$y_i = c_1 e^{-x_i} + c_2 x_i$$

where $x_i = 0.1i$ for $i = 1 \ldots 10$, $c_1 = 5$, and $c_2 = 4$. Noise is added to y_i and the coefficients c_1 and c_2 are estimated using linear least squares.

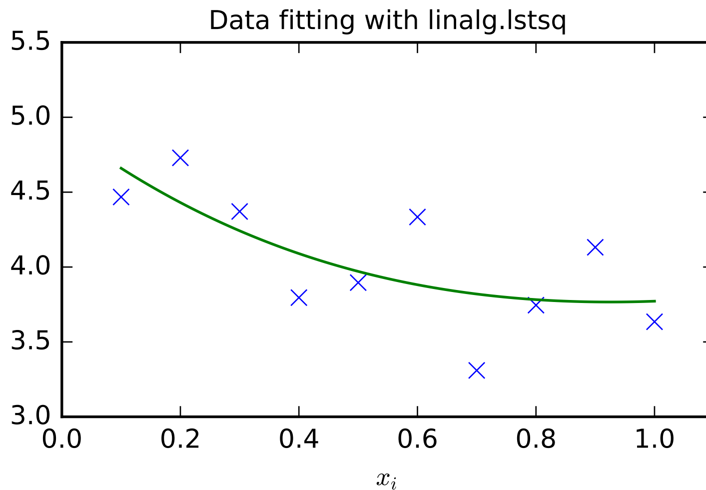
```
>>> import numpy as np
>>> from scipy import linalg
>>> import matplotlib.pyplot as plt
```

```
>>> c1, c2 = 5.0, 2.0
>>> i = np.r_[1:11]
>>> xi = 0.1*i
>>> yi = c1*np.exp(-xi) + c2*xi
>>> zi = yi + 0.05 * np.max(yi) * np.random.randn(len(yi))
```

```
>>> A = np.c_[np.exp(-xi)[:, np.newaxis], xi[:, np.newaxis]]
>>> c, resid, rank, sigma = linalg.lstsq(A, zi)
```

```
>>> xi2 = np.r_[0.1:1.0:100j]
>>> yi2 = c[0]*np.exp(-xi2) + c[1]*xi2
```

```
>>> plt.plot(xi,zi,'x',xi2,yi2)
>>> plt.axis([0,1.1,3.0,5.5])
>>> plt.xlabel('$x_i$')
>>> plt.title('Data fitting with linalg.lstsq')
>>> plt.show()
```



Generalized inverse

The generalized inverse is calculated using the command `linalg.pinv` or `linalg.pinv2`. These two commands differ in how they compute the generalized inverse. The first uses the linalg.lstsq algorithm while the second uses singular value decomposition. Let $\mathbf{A}$ be an $M \times N$ matrix, then if $M > N$ the generalized inverse is

$$\mathbf{A}^{\dagger} = \left(\mathbf{A}^{H}\mathbf{A}\right)^{-1}\mathbf{A}^{H}$$

while if $M < N$ matrix the generalized inverse is

$$\mathbf{A}^{\#} = \mathbf{A}^{H}\left(\mathbf{A}\mathbf{A}^{H}\right)^{-1}.$$

In both cases for $M = N$, then

$$\mathbf{A}^{\dagger} = \mathbf{A}^{\#} = \mathbf{A}^{-1}$$

as long as $\mathbf{A}$ is invertible.

Decompositions

In many applications it is useful to decompose a matrix using other representations. There are several decompositions supported by SciPy.

Eigenvalues and eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. In one popular form, the eigenvalue-eigenvector problem is to find for some square matrix $\mathbf{A}$ scalars λ and corresponding vectors $\mathbf{v}$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

For an $N \times N$ matrix, there are N (not necessarily distinct) eigenvalues — roots of the (characteristic) polynomial

$$|\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The eigenvectors, $\mathbf{v}$, are also sometimes called right eigenvectors to distinguish them from another set of left eigenvectors that satisfy

$$\mathbf{v}_L^H\mathbf{A} = \lambda\mathbf{v}_L^H$$

or

$$\mathbf{A}^H \mathbf{v}_L = \lambda^* \mathbf{v}_L.$$

With it's default optional arguments, the command `linalg.eig` returns λ and $\mathbf{v}$. However, it can also return $\mathbf{v}_L$ and just λ by itself (`linalg.eigvals` returns just λ as well).

In addition, `linalg.eig` can also solve the more general eigenvalue problem

$$\begin{aligned}\mathbf{A}\mathbf{v} &= \lambda \mathbf{B}\mathbf{v} \\ \mathbf{A}^H \mathbf{v}_L &= \lambda^* \mathbf{B}^H \mathbf{v}_L\end{aligned}$$

for square matrices $\mathbf{A}$ and $\mathbf{B}$. The standard eigenvalue problem is an example of the general eigenvalue problem for $\mathbf{B} = \mathbf{I}$. When a generalized eigenvalue problem can be solved, then it provides a decomposition of $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{B}\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$$

where $\mathbf{V}$ is the collection of eigenvectors into columns and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues.

By definition, eigenvectors are only defined up to a constant scale factor. In SciPy, the scaling factor for the eigenvectors is chosen so that $\|\mathbf{v}\|^2 = \sum_i v_i^2 = 1$.

As an example, consider finding the eigenvalues and eigenvectors of the matrix

$$\mathbf{A} = \left[\begin{array}{ccc} 1 & 5 & 2 \\ 2 & 4 & 1 \\ 3 & 6 & 2 \end{array}\right].$$

The characteristic polynomial is

$$\begin{aligned}|\mathbf{A} - \lambda \mathbf{I}| &= (1-\lambda)\left[(4-\lambda)(2-\lambda) - 6\right] - \\ & \quad 5\left[2(2-\lambda) - 3\right] + 2\left[12 - 3(4-\lambda)\right] \\ &= -\lambda^3 + 7\lambda^2 + 8\lambda - 3.\end{aligned}$$

The roots of this polynomial are the eigenvalues of $\mathbf{A}$:

$$\begin{aligned}\lambda_1 &= 7.9579 \\ \lambda_2 &= -1.2577 \\ \lambda_3 &= 0.2997.\end{aligned}$$

The eigenvectors corresponding to each eigenvalue can be found using the original equation. The eigenvectors associated with these eigenvalues can then be found.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> la, v = linalg.eig(A)
>>> l1, l2 = la
>>> print l1, l2   # eigenvalues
(-0.372281323269+0j) (5.37228132327+0j)
>>> print v[:, 0]   # first eigenvector
[-0.82456484  0.56576746]
>>> print v[:, 1]   # second eigenvector
[-0.41597356 -0.90937671]
>>> print np.sum(abs(v**2), axis=0)  # eigenvectors are unitary
[ 1.  1.]
>>> v1 = np.array(v[:, 0]).T
>>> print linalg.norm(A.dot(v1) - l1*v1)  # check the computation
3.23682852457e-16
```

Singular value decomposition

Singular Value Decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square. Let $\mathbf{A}$ be an $M\times N$ matrix with M and N arbitrary. The matrices $\mathbf{A}^{H}\mathbf{A}$ and $\mathbf{A}\mathbf{A}^{H}$ are square hermitian matrices[1] of size $N\times N$ and $M\times M$ respectively. It is known that the eigenvalues of square hermitian matrices are real and non-negative. In addition, there are at most $\min\left(M,N\right)$ identical non-zero eigenvalues of $\mathbf{A}^{H}\mathbf{A}$ and $\mathbf{A}\mathbf{A}^{H}$. Define these positive eigenvalues as σ_{i}^{2}. The square-root of these are called singular values of $\mathbf{A}$. The eigenvectors of $\mathbf{A}^{H}\mathbf{A}$ are collected by columns into an $N\times N$ unitary[2] matrix $\mathbf{V}$ while the eigenvectors of $\mathbf{A}\mathbf{A}^{H}$ are collected by columns in the unitary matrix $\mathbf{U}$, the singular values are collected in an $M\times N$ zero matrix $\mathbf{\Sigma}$ with main diagonal entries set to the singular values. Then

$$\mathbf{A}=\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{H}$$

is the singular-value decomposition of $\mathbf{A}$. Every matrix has a singular value decomposition. Sometimes, the singular values are called the spectrum of $\mathbf{A}$. The command `linalg.svd` will return $\mathbf{U}$, $\mathbf{V}^{H}$, and σ_{i} as an array of the singular values. To obtain the matrix $\mathbf{\Sigma}$ use `linalg.diagsvd`. The following example illustrates the use of `linalg.svd`.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2,3],[4,5,6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> M,N = A.shape
>>> U,s,Vh = linalg.svd(A)
>>> Sig = linalg.diagsvd(s,M,N)
>>> U, Vh = U, Vh
>>> U
array([[-0.3863177 , -0.92236578],
       [-0.92236578,  0.3863177 ]])
>>> Sig
array([[ 9.508032  ,  0.        ,  0.        ],
       [ 0.        ,  0.77286964,  0.        ]])
>>> Vh
array([[-0.42866713, -0.56630692, -0.7039467 ],
       [ 0.80596391,  0.11238241, -0.58119908],
       [ 0.40824829, -0.81649658,  0.40824829]])
>>> U.dot(Sig.dot(Vh)) #check computation
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

LU decomposition

The LU decomposition finds a representation for the $M\times N$ matrix $\mathbf{A}$ as

$$\mathbf{A}=\mathbf{P}\,\mathbf{L}\,\mathbf{U}$$

where $\mathbf{P}$ is an $M\times M$ permutation matrix (a permutation of the rows of the identity matrix), $\mathbf{L}$ is in $M\times K$ lower triangular or trapezoidal matrix ($K=\min\left(M,N\right)$) with unit-diagonal, and $\mathbf{U}$ is an upper triangular or trapezoidal matrix. The SciPy command for this decomposition is `linalg.lu`.

Such a decomposition is often useful for solving many simultaneous equations where the left-hand-side does not change but the right hand side does. For example, suppose we are going to solve

$$\mathbf{A}\mathbf{x}_{i}=\mathbf{b}_{i}$$

[1] A hermitian matrix $\mathbf{D}$ satisfies $\mathbf{D}^{H}=\mathbf{D}$.

[2] A unitary matrix $\mathbf{D}$ satisfies $\mathbf{D}^{H}\mathbf{D}=\mathbf{I}=\mathbf{D}\mathbf{D}^{H}$ so that $\mathbf{D}^{-1}=\mathbf{D}^{H}$.

for many different $\mathbf{b}_i$. The LU decomposition allows this to be written as

$$\mathbf{PLUx}_i = \mathbf{b}_i.$$

Because $\mathbf{L}$ is lower-triangular, the equation can be solved for $\mathbf{Ux}_i$ and finally $\mathbf{x}_i$ very rapidly using forward- and back-substitution. An initial time spent factoring $\mathbf{A}$ allows for very rapid solution of similar systems of equations in the future. If the intent for performing LU decomposition is for solving linear systems then the command `linalg.lu_factor` should be used followed by repeated applications of the command `linalg.lu_solve` to solve the system for each new right-hand-side.

Cholesky decomposition

Cholesky decomposition is a special case of LU decomposition applicable to Hermitian positive definite matrices. When $\mathbf{A} = \mathbf{A}^H$ and $\mathbf{x}^H\mathbf{Ax} \geq 0$ for all $\mathbf{x}$, then decompositions of $\mathbf{A}$ can be found so that

$$\begin{aligned} \mathbf{A} &= \mathbf{U}^H\mathbf{U} \\ \mathbf{A} &= \mathbf{LL}^H \end{aligned}$$

where $\mathbf{L}$ is lower-triangular and $\mathbf{U}$ is upper triangular. Notice that $\mathbf{L} = \mathbf{U}^H$. The command `linalg.cholesky` computes the cholesky factorization. For using cholesky factorization to solve systems of equations there are also `linalg.cho_factor` and `linalg.cho_solve` routines that work similarly to their LU decomposition counterparts.

QR decomposition

The QR decomposition (sometimes called a polar decomposition) works for any $M \times N$ array and finds an $M \times M$ unitary matrix $\mathbf{Q}$ and an $M \times N$ upper-trapezoidal matrix $\mathbf{R}$ such that

$$\mathbf{A} = \mathbf{QR}.$$

Notice that if the SVD of $\mathbf{A}$ is known then the QR decomposition can be found

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \mathbf{QR}$$

implies that $\mathbf{Q} = \mathbf{U}$ and $\mathbf{R} = \boldsymbol{\Sigma}\mathbf{V}^H$. Note, however, that in SciPy independent algorithms are used to find QR and SVD decompositions. The command for QR decomposition is `linalg.qr` .

Schur decomposition

For a square $N \times N$ matrix, $\mathbf{A}$, the Schur decomposition finds (not-necessarily unique) matrices $\mathbf{T}$ and $\mathbf{Z}$ such that

$$\mathbf{A} = \mathbf{ZTZ}^H$$

where $\mathbf{Z}$ is a unitary matrix and $\mathbf{T}$ is either upper-triangular or quasi-upper triangular depending on whether or not a real schur form or complex schur form is requested. For a real schur form both $\mathbf{T}$ and $\mathbf{Z}$ are real-valued when $\mathbf{A}$ is real-valued. When $\mathbf{A}$ is a real-valued matrix the real schur form is only quasi-upper triangular because 2×2 blocks extrude from the main diagonal corresponding to any complex- valued eigenvalues. The command `linalg.schur` finds the Schur decomposition while the command `linalg.rsf2csf` converts $\mathbf{T}$ and $\mathbf{Z}$ from a real Schur form to a complex Schur form. The Schur form is especially useful in calculating functions of matrices.

The following example illustrates the schur decomposition:

```
>>> from scipy import linalg
>>> A = np.mat('[1 3 2; 1 4 5; 2 3 6]')
>>> T, Z = linalg.schur(A)
>>> T1, Z1 = linalg.schur(A, 'complex')
>>> T2, Z2 = linalg.rsf2csf(T, Z)
>>> T
```

```
array([[ 9.90012467,  1.78947961, -0.65498528],
       [ 0.        ,  0.54993766, -1.57754789],
       [ 0.        ,  0.51260928,  0.54993766]])
>>> T2
array([[ 9.90012467 +0.00000000e+00j, -0.32436598 +1.55463542e+00j,
        -0.88619748 +5.69027615e-01j],
       [ 0.00000000 +0.00000000e+00j,  0.54993766 +8.99258408e-01j,
         1.06493862 -5.80496735e-16j],
       [ 0.00000000 +0.00000000e+00j,  0.00000000 +0.00000000e+00j,
         0.54993766 -8.99258408e-01j]])
>>> abs(T1 - T2) # different
array([[  1.06604538e-14,   2.06969555e+00,   1.69375747e+00],  # may vary
       [  0.00000000e+00,   1.33688556e-15,   4.74146496e-01],
       [  0.00000000e+00,   0.00000000e+00,   1.13220977e-15]])
>>> abs(Z1 - Z2) # different
array([[ 0.06833781,  0.88091091,  0.79568503],    # may vary
       [ 0.11857169,  0.44491892,  0.99594171],
       [ 0.12624999,  0.60264117,  0.77257633]])
>>> T, Z, T1, Z1, T2, Z2 = map(np.mat,(T,Z,T1,Z1,T2,Z2))
>>> abs(A - Z*T*Z.H)  # same
matrix([[  5.55111512e-16,   1.77635684e-15,   2.22044605e-15],
        [  0.00000000e+00,   3.99680289e-15,   8.88178420e-16],
        [  1.11022302e-15,   4.44089210e-16,   3.55271368e-15]])
>>> abs(A - Z1*T1*Z1.H)  # same
matrix([[  4.26993904e-15,   6.21793362e-15,   8.00007092e-15],
        [  5.77945386e-15,   6.21798014e-15,   1.06653681e-14],
        [  7.16681444e-15,   8.90271058e-15,   1.77635764e-14]])
>>> abs(A - Z2*T2*Z2.H)  # same
matrix([[  6.02594127e-16,   1.77648931e-15,   2.22506907e-15],
        [  2.46275555e-16,   3.99684548e-15,   8.91642616e-16],
        [  8.88225111e-16,   8.88312432e-16,   4.44104848e-15]])
```

Interpolative Decomposition

`scipy.linalg.interpolative` contains routines for computing the interpolative decomposition (ID) of a matrix. For a matrix $A \in \mathbb{C}^{m \times n}$ of rank $k \leq \min\{m, n\}$ this is a factorization

$$A\Pi = \begin{bmatrix} A\Pi_1 & A\Pi_2 \end{bmatrix} = A\Pi_1 \begin{bmatrix} I & T \end{bmatrix},$$

where $\Pi = [\Pi_1, \Pi_2]$ is a permutation matrix with $\Pi_1 \in \{0, 1\}^{n \times k}$, i.e., $A\Pi_2 = A\Pi_1 T$. This can equivalently be written as $A = BP$, where $B = A\Pi_1$ and $P = [I, T]\Pi^\mathsf{T}$ are the *skeleton* and *interpolation matrices*, respectively.

See also:

`scipy.linalg.interpolative` — for more information.

Matrix Functions

Consider the function $f(x)$ with Taylor series expansion

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k.$$

A matrix function can be defined using this Taylor series for the square matrix $\mathbf{A}$ as

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} \mathbf{A}^k.$$

While, this serves as a useful representation of a matrix function, it is rarely the best way to calculate a matrix function.

Exponential and logarithm functions

The matrix exponential is one of the more common matrix functions. It can be defined for square matrices as

$$e^{\mathbf{A}}=\sum_{k=0}^{\infty}\frac{1}{k!}\mathbf{A}^{k}.$$

The command `linalg.expm3` uses this Taylor series definition to compute the matrix exponential. Due to poor convergence properties it is not often used.

Another method to compute the matrix exponential is to find an eigenvalue decomposition of $\mathbf{A}$:

$$\mathbf{A}=\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1}$$

and note that

$$e^{\mathbf{A}}=\mathbf{V}e^{\boldsymbol{\Lambda}}\mathbf{V}^{-1}$$

where the matrix exponential of the diagonal matrix $\boldsymbol{\Lambda}$ is just the exponential of its elements. This method is implemented in `linalg.expm2` .

The preferred method for implementing the matrix exponential is to use scaling and a Padé approximation for e^{x} . This algorithm is implemented as `linalg.expm` .

The inverse of the matrix exponential is the matrix logarithm defined as the inverse of the matrix exponential.

$$\mathbf{A}\equiv\exp\left(\log\left(\mathbf{A}\right)\right).$$

The matrix logarithm can be obtained with `linalg.logm` .

Trigonometric functions

The trigonometric functions $\sin$, $\cos$, and $\tan$ are implemented for matrices in `linalg.sinm`, `linalg.cosm`, and `linalg.tanm` respectively. The matrix sin and cosine can be defined using Euler's identity as

$$\begin{aligned}\sin\left(\mathbf{A}\right) &= \frac{e^{j\mathbf{A}}-e^{-j\mathbf{A}}}{2j}\\ \cos\left(\mathbf{A}\right) &= \frac{e^{j\mathbf{A}}+e^{-j\mathbf{A}}}{2}.\end{aligned}$$

The tangent is

$$\tan\left(x\right)=\frac{\sin\left(x\right)}{\cos\left(x\right)}=\left[\cos\left(x\right)\right]^{-1}\sin\left(x\right)$$

and so the matrix tangent is defined as

$$\left[\cos\left(\mathbf{A}\right)\right]^{-1}\sin\left(\mathbf{A}\right).$$

Hyperbolic trigonometric functions

The hyperbolic trigonometric functions $\sinh$, $\cosh$, and $\tanh$ can also be defined for matrices using the familiar definitions:

$$\begin{aligned}\sinh\left(\mathbf{A}\right) &= \frac{e^{\mathbf{A}}-e^{-\mathbf{A}}}{2}\\ \cosh\left(\mathbf{A}\right) &= \frac{e^{\mathbf{A}}+e^{-\mathbf{A}}}{2}\\ \tanh\left(\mathbf{A}\right) &= \left[\cosh\left(\mathbf{A}\right)\right]^{-1}\sinh\left(\mathbf{A}\right).\end{aligned}$$

These matrix functions can be found using `linalg.sinhm`, `linalg.coshm` , and `linalg.tanhm`.

Arbitrary function

Finally, any arbitrary function that takes one complex number and returns a complex number can be called as a matrix function using the command `linalg.funm`. This command takes the matrix and an arbitrary Python function. It then implements an algorithm from Golub and Van Loan's book "Matrix Computations" to compute the function applied to the matrix using a Schur decomposition. Note that *the function needs to accept complex numbers* as input in order to work with this algorithm. For example the following code computes the zeroth-order Bessel function applied to a matrix.

```
>>> from scipy import special, random, linalg
>>> np.random.seed(1234)
>>> A = random.rand(3, 3)
>>> B = linalg.funm(A, lambda x: special.jv(0, x))
>>> A
array([[ 0.19151945,  0.62210877,  0.43772774],
       [ 0.78535858,  0.77997581,  0.27259261],
       [ 0.27646426,  0.80187218,  0.95813935]])
>>> B
array([[ 0.86511146, -0.19676526, -0.13856748],
       [-0.17479869,  0.7259118 , -0.16606258],
       [-0.19212044, -0.32052767,  0.73590704]])
>>> linalg.eigvals(A)
array([ 1.73881510+0.j, -0.20270676+0.j,  0.39352627+0.j])
>>> special.jv(0, linalg.eigvals(A))
array([ 0.37551908+0.j,  0.98975384+0.j,  0.96165739+0.j])
>>> linalg.eigvals(B)
array([ 0.37551908+0.j,  0.98975384+0.j,  0.96165739+0.j])
```

Note how, by virtue of how matrix analytic functions are defined, the Bessel function has acted on the matrix eigenvalues.

Special matrices

SciPy and NumPy provide several functions for creating special matrices that are frequently used in engineering and science.

Type	Function	Description
block diagonal	*scipy.linalg.block_diag*	Create a block diagonal matrix from the provided arrays.
circulant	*scipy.linalg.circulant*	Construct a circulant matrix.
companion	*scipy.linalg.companion*	Create a companion matrix.
Hadamard	*scipy.linalg.hadamard*	Construct a Hadamard matrix.
Hankel	*scipy.linalg.hankel*	Construct a Hankel matrix.
Hilbert	*scipy.linalg.hilbert*	Construct a Hilbert matrix.
Inverse Hilbert	*scipy.linalg.invhilbert*	Construct the inverse of a Hilbert matrix.
Leslie	*scipy.linalg.leslie*	Create a Leslie matrix.
Pascal	*scipy.linalg.pascal*	Create a Pascal matrix.
Toeplitz	*scipy.linalg.toeplitz*	Construct a Toeplitz matrix.
Van der Monde	`numpy.vander`	Generate a Van der Monde matrix.

For examples of the use of these functions, see their respective docstrings.

3.1.10 Sparse Eigenvalue Problems with ARPACK

Introduction

ARPACK is a Fortran package which provides routines for quickly finding a few eigenvalues/eigenvectors of large sparse matrices. In order to find these solutions, it requires only left-multiplication by the matrix in question. This operation is performed through a *reverse-communication* interface. The result of this structure is that ARPACK is able to find eigenvalues and eigenvectors of any linear function mapping a vector to a vector.

All of the functionality provided in ARPACK is contained within the two high-level interfaces `scipy.sparse.linalg.eigs` and `scipy.sparse.linalg.eigsh`. `eigs` provides interfaces to find the eigenvalues/vectors of real or complex nonsymmetric square matrices, while `eigsh` provides interfaces for real-symmetric or complex-hermitian matrices.

Basic Functionality

ARPACK can solve either standard eigenvalue problems of the form

$$A\mathbf{x} = \lambda\mathbf{x}$$

or general eigenvalue problems of the form

$$A\mathbf{x} = \lambda M\mathbf{x}$$

The power of ARPACK is that it can compute only a specified subset of eigenvalue/eigenvector pairs. This is accomplished through the keyword `which`. The following values of `which` are available:

- `which = 'LM'` : Eigenvalues with largest magnitude (`eigs`, `eigsh`), that is, largest eigenvalues in the euclidean norm of complex numbers.
- `which = 'SM'` : Eigenvalues with smallest magnitude (`eigs`, `eigsh`), that is, smallest eigenvalues in the euclidean norm of complex numbers.
- `which = 'LR'` : Eigenvalues with largest real part (`eigs`)
- `which = 'SR'` : Eigenvalues with smallest real part (`eigs`)
- `which = 'LI'` : Eigenvalues with largest imaginary part (`eigs`)
- `which = 'SI'` : Eigenvalues with smallest imaginary part (`eigs`)
- `which = 'LA'` : Eigenvalues with largest algebraic value (`eigsh`), that is, largest eigenvalues inclusive of any negative sign.
- `which = 'SA'` : Eigenvalues with smallest algebraic value (`eigsh`), that is, smallest eigenvalues inclusive of any negative sign.
- `which = 'BE'` : Eigenvalues from both ends of the spectrum (`eigsh`)

Note that ARPACK is generally better at finding extremal eigenvalues: that is, eigenvalues with large magnitudes. In particular, using `which = 'SM'` may lead to slow execution time and/or anomalous results. A better approach is to use *shift-invert mode*.

Shift-Invert Mode

Shift invert mode relies on the following observation. For the generalized eigenvalue problem

$$A\mathbf{x} = \lambda M\mathbf{x}$$

it can be shown that

$$(A - \sigma M)^{-1} M \mathbf{x} = \nu \mathbf{x}$$

where

$$\nu = \frac{1}{\lambda - \sigma}$$

Examples

Imagine you'd like to find the smallest and largest eigenvalues and the corresponding eigenvectors for a large matrix. ARPACK can handle many forms of input: dense matrices such as `numpy.ndarray` instances, sparse matrices such as *scipy.sparse.csr_matrix*, or a general linear operator derived from *scipy.sparse.linalg.LinearOperator*. For this example, for simplicity, we'll construct a symmetric, positive-definite matrix.

```
>>> import numpy as np
>>> from scipy.linalg import eigh
>>> from scipy.sparse.linalg import eigsh
>>> np.set_printoptions(suppress=True)
>>>
>>> np.random.seed(0)
>>> X = np.random.random((100,100)) - 0.5
>>> X = np.dot(X, X.T) #create a symmetric matrix
```

We now have a symmetric matrix `X` with which to test the routines. First compute a standard eigenvalue decomposition using `eigh`:

```
>>> evals_all, evecs_all = eigh(X)
```

As the dimension of `X` grows, this routine becomes very slow. Especially if only a few eigenvectors and eigenvalues are needed, `ARPACK` can be a better option. First let's compute the largest eigenvalues (`which = 'LM'`) of `X` and compare them to the known results:

```
>>> evals_large, evecs_large = eigsh(X, 3, which='LM')
>>> print evals_all[-3:]
[ 29.1446102   30.05821805  31.19467646]
>>> print evals_large
[ 29.1446102   30.05821805  31.19467646]
>>> print np.dot(evecs_large.T, evecs_all[:,-3:])
array([[-1.  0.  0.],       # may vary (signs)
       [ 0.  1.  0.],
       [-0.  0. -1.]])
```

The results are as expected. ARPACK recovers the desired eigenvalues, and they match the previously known results. Furthermore, the eigenvectors are orthogonal, as we'd expect. Now let's attempt to solve for the eigenvalues with smallest magnitude:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM')
Traceback (most recent call last):       # may vary (convergence)
...
scipy.sparse.linalg.eigen.arpack.arpack.ArpackNoConvergence:
ARPACK error -1: No convergence (1001 iterations, 0/3 eigenvectors converged)
```

Oops. We see that as mentioned above, `ARPACK` is not quite as adept at finding small eigenvalues. There are a few ways this problem can be addressed. We could increase the tolerance (`tol`) to lead to faster convergence:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM', tol=1E-2)
>>> evals_all[:3]
array([0.0003783, 0.00122714, 0.00715878])
>>> evals_small
array([0.00037831, 0.00122714, 0.00715881])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 0.99999999  0.00000024 -0.00000049],    # may vary (signs)
       [-0.00000023  0.99999999  0.00000056],
       [ 0.00000031 -0.00000037  0.99999852]])
```

This works, but we lose the precision in the results. Another option is to increase the maximum number of iterations (`maxiter`) from 1000 to 5000:

```
>>> evals_small, evecs_small = eigsh(X, 3, which='SM', maxiter=5000)
>>> evals_all[:3]
array([0.0003783, 0.00122714, 0.00715878])
>>> evals_small
array([0.0003783, 0.00122714, 0.00715878])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 1.  0.  0.],           # may vary (signs)
       [-0.  1.  0.],
       [ 0.  0. -1.]])
```

We get the results we'd hoped for, but the computation time is much longer. Fortunately, `ARPACK` contains a mode that allows quick determination of non-external eigenvalues: *shift-invert mode*. As mentioned above, this mode involves transforming the eigenvalue problem to an equivalent problem with different eigenvalues. In this case, we hope to find eigenvalues near zero, so we'll choose `sigma = 0`. The transformed eigenvalues will then satisfy $\nu = 1/(\sigma - \lambda) = 1/\lambda$, so our small eigenvalues λ become large eigenvalues ν.

```
>>> evals_small, evecs_small = eigsh(X, 3, sigma=0, which='LM')
>>> evals_all[:3]
array([0.0003783, 0.00122714, 0.00715878])
>>> evals_small
array([0.0003783, 0.00122714, 0.00715878])
>>> np.dot(evecs_small.T, evecs_all[:,:3])
array([[ 1.  0.  0.],    # may vary (signs)
       [ 0. -1. -0.],
       [-0. -0.  1.]])
```

We get the results we were hoping for, with much less computational time. Note that the transformation from $\nu \rightarrow \lambda$ takes place entirely in the background. The user need not worry about the details.

The shift-invert mode provides more than just a fast way to obtain a few small eigenvalues. Say you desire to find internal eigenvalues and eigenvectors, e.g. those nearest to $\lambda = 1$. Simply set `sigma = 1` and ARPACK takes care of the rest:

```
>>> evals_mid, evecs_mid = eigsh(X, 3, sigma=1, which='LM')
>>> i_sort = np.argsort(abs(1. / (1 - evals_all)))[-3:]
>>> evals_all[i_sort]
array([1.16577199, 0.85081388, 1.06642272])
>>> evals_mid
array([0.85081388, 1.06642272, 1.16577199])
>>> print np.dot(evecs_mid.T, evecs_all[:,i_sort])
array([[-0.  1.  0.],     # may vary (signs)
       [-0. -0.  1.],
       [ 1.  0.  0.]]
```

The eigenvalues come out in a different order, but they're all there. Note that the shift-invert mode requires the internal solution of a matrix inverse. This is taken care of automatically by `eigsh` and *eigs*, but the operation can also be specified by the user. See the docstring of *scipy.sparse.linalg.eigsh* and *scipy.sparse.linalg.eigs* for details.

References

3.1.11 Compressed Sparse Graph Routines (scipy.sparse.csgraph)

Example: Word Ladders

A Word Ladder is a word game invented by Lewis Carroll in which players find paths between words by switching one letter at a time. For example, one can link "ape" and "man" in the following way:

$$\text{ape} \rightarrow \text{apt} \rightarrow \text{ait} \rightarrow \text{bit} \rightarrow \text{big} \rightarrow \text{bag} \rightarrow \text{mag} \rightarrow \text{man}$$

Note that each step involves changing just one letter of the word. This is just one possible path from "ape" to "man", but is it the shortest possible path? If we desire to find the shortest word ladder path between two given words, the sparse graph submodule can help.

First we need a list of valid words. Many operating systems have such a list built-in. For example, on linux, a word list can often be found at one of the following locations:

```
/usr/share/dict
/var/lib/dict
```

Another easy source for words are the scrabble word lists available at various sites around the internet (search with your favorite search engine). We'll first create this list. The system word lists consist of a file with one word per line. The following should be modified to use the particular word list you have available:

```
>>> word_list = open('/usr/share/dict/words').readlines()
>>> word_list = map(str.strip, word_list)
```

We want to look at words of length 3, so let's select just those words of the correct length. We'll also eliminate words which start with upper-case (proper nouns) or contain non alpha-numeric characters like apostrophes and hyphens. Finally, we'll make sure everything is lower-case for comparison later:

```
>>> word_list = [word for word in word_list if len(word) == 3]
>>> word_list = [word for word in word_list if word[0].islower()]
>>> word_list = [word for word in word_list if word.isalpha()]
>>> word_list = map(str.lower, word_list)
>>> len(word_list)
586
```

Now we have a list of 586 valid three-letter words (the exact number may change depending on the particular list used). Each of these words will become a node in our graph, and we will create edges connecting the nodes associated with each pair of words which differs by only one letter.

There are efficient ways to do this, and inefficient ways to do this. To do this as efficiently as possible, we're going to use some sophisticated numpy array manipulation:

```
>>> import numpy as np
>>> word_list = np.asarray(word_list)
>>> word_list.dtype
dtype('|S3')
>>> word_list.sort()  # sort for quick searching later
```

We have an array where each entry is three bytes. We'd like to find all pairs where exactly one byte is different. We'll start by converting each word to a three-dimensional vector:

```
>>> word_bytes = np.ndarray((word_list.size, word_list.itemsize),
...                         dtype='int8',
...                         buffer=word_list.data)
>>> word_bytes.shape
(586, 3)
```

Now we'll use the Hamming distance between each point to determine which pairs of words are connected. The Hamming distance measures the fraction of entries between two vectors which differ: any two words with a hamming distance equal to $1/N$, where N is the number of letters, are connected in the word ladder:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> from scipy.sparse import csr_matrix
>>> hamming_dist = pdist(word_bytes, metric='hamming')
>>> graph = csr_matrix(squareform(hamming_dist < 1.5 / word_list.itemsize))
```

When comparing the distances, we don't use an equality because this can be unstable for floating point values. The inequality produces the desired result as long as no two entries of the word list are identical. Now that our graph is set up, we'll use a shortest path search to find the path between any two words in the graph:

```
>>> i1 = word_list.searchsorted('ape')
>>> i2 = word_list.searchsorted('man')
>>> word_list[i1]
'ape'
>>> word_list[i2]
'man'
```

We need to check that these match, because if the words are not in the list that will not be the case. Now all we need is to find the shortest path between these two indices in the graph. We'll use Dijkstra's algorithm, because it allows us to find the path for just one node:

```
>>> from scipy.sparse.csgraph import dijkstra
>>> distances, predecessors = dijkstra(graph, indices=i1,
...                                    return_predecessors=True)
>>> print distances[i2]
5.0
```

So we see that the shortest path between 'ape' and 'man' contains only five steps. We can use the predecessors returned by the algorithm to reconstruct this path:

```
>>> path = []
>>> i = i2
>>> while i != i1:
...     path.append(word_list[i])
...     i = predecessors[i]
>>> path.append(word_list[i1])
>>> print path[::-1]
['ape', 'apt', 'opt', 'oat', 'mat', 'man']
```

This is three fewer links than our initial example: the path from ape to man is only five steps.

Using other tools in the module, we can answer other questions. For example, are there three-letter words which are not linked in a word ladder? This is a question of connected components in the graph:

```
>>> from scipy.sparse.csgraph import connected_components
>>> N_components, component_list = connected_components(graph)
```

```
>>> print N_components
15
```

In this particular sample of three-letter words, there are 15 connected components: that is, 15 distinct sets of words with no paths between the sets. How many words are in each of these sets? We can learn this from the list of components:

```
>>> [np.sum(component_list == i) for i in range(15)]
[571, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

There is one large connected set, and 14 smaller ones. Let's look at the words in the smaller ones:

```
>>> [list(word_list[np.where(component_list == i)]) for i in range(1, 15)]
[['aha'],
 ['chi'],
 ['ebb'],
 ['ems', 'emu'],
 ['gnu'],
 ['ism'],
 ['khz'],
 ['nth'],
 ['ova'],
 ['qua'],
 ['ugh'],
 ['ups'],
 ['urn'],
 ['use']]
```

These are all the three-letter words which do not connect to others via a word ladder.

We might also be curious about which words are maximally separated. Which two words take the most links to connect? We can determine this by computing the matrix of all shortest paths. Note that by convention, the distance between two non-connected points is reported to be infinity, so we'll need to remove these before finding the maximum:

```
>>> distances, predecessors = dijkstra(graph, return_predecessors=True)
>>> np.max(distances[~np.isinf(distances)])
13.0
```

So there is at least one pair of words which takes 13 steps to get from one to the other! Let's determine which these are:

```
>>> i1, i2 = np.where(distances == 13)
>>> zip(word_list[i1], word_list[i2])
[('imp', 'ohm'),
 ('imp', 'ohs'),
 ('ohm', 'imp'),
 ('ohm', 'ump'),
 ('ohs', 'imp'),
 ('ohs', 'ump'),
 ('ump', 'ohm'),
 ('ump', 'ohs')]
```

We see that there are two pairs of words which are maximally separated from each other: 'imp' and 'ump' on one hand, and 'ohm' and 'ohs' on the other hand. We can find the connecting list in the same way as above:

```
>>> path = []
>>> i = i2[0]
>>> while i != i1[0]:
...     path.append(word_list[i])
```

```
...     i = predecessors[i1[0], i]
>>> path.append(word_list[i1[0]])
>>> print path[::-1]
['imp', 'amp', 'asp', 'ass', 'ads', 'add', 'aid', 'mid', 'mod', 'moo', 'too', 'tho',
↪'oho', 'ohm']
```

This gives us the path we desired to see.

Word ladders are just one potential application of scipy's fast graph algorithms for sparse matrices. Graph theory makes appearances in many areas of mathematics, data analysis, and machine learning. The sparse graph tools are flexible enough to handle many of these situations.

3.1.12 Spatial data structures and algorithms (`scipy.spatial`)

scipy.spatial can compute triangulations, Voronoi diagrams, and convex hulls of a set of points, by leveraging the Qhull library.

Moreover, it contains *KDTree* implementations for nearest-neighbor point queries, and utilities for distance computations in various metrics.

Delaunay triangulations

The Delaunay triangulation is a subdivision of a set of points into a non-overlapping set of triangles, such that no point is inside the circumcircle of any triangle. In practice, such triangulations tend to avoid triangles with small angles.

Delaunay triangulation can be computed using *scipy.spatial* as follows:

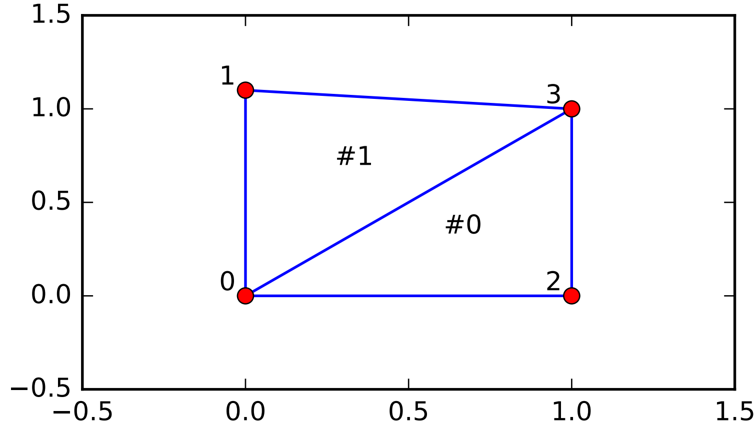
```
>>> from scipy.spatial import Delaunay
>>> points = np.array([[0, 0], [0, 1.1], [1, 0], [1, 1]])
>>> tri = Delaunay(points)
```

We can visualize it:

```
>>> import matplotlib.pyplot as plt
>>> plt.triplot(points[:,0], points[:,1], tri.simplices.copy())
>>> plt.plot(points[:,0], points[:,1], 'o')
```

And add some further decorations:

```
>>> for j, p in enumerate(points):
...     plt.text(p[0]-0.03, p[1]+0.03, j, ha='right') # label the points
>>> for j, s in enumerate(tri.simplices):
...     p = points[s].mean(axis=0)
...     plt.text(p[0], p[1], '#%d' % j, ha='center') # label triangles
>>> plt.xlim(-0.5, 1.5); plt.ylim(-0.5, 1.5)
>>> plt.show()
```



The structure of the triangulation is encoded in the following way: the `simplices` attribute contains the indices of the points in the `points` array that make up the triangle. For instance:

```
>>> i = 1
>>> tri.simplices[i,:]
array([3, 1, 0], dtype=int32)
>>> points[tri.simplices[i,:]]
array([[ 1. ,  1. ],
       [ 0. ,  1.1],
       [ 0. ,  0. ]])
```

Moreover, neighboring triangles can also be found out:

```
>>> tri.neighbors[i]
array([-1,  0, -1], dtype=int32)
```

What this tells us is that this triangle has triangle #0 as a neighbor, but no other neighbors. Moreover, it tells us that neighbor 0 is opposite the vertex 1 of the triangle:

```
>>> points[tri.simplices[i, 1]]
array([ 0. ,  1.1])
```

Indeed, from the figure we see that this is the case.

Qhull can also perform tesselations to simplices also for higher-dimensional point sets (for instance, subdivision into tetrahedra in 3-D).

Coplanar points

It is important to note that not *all* points necessarily appear as vertices of the triangulation, due to numerical precision issues in forming the triangulation. Consider the above with a duplicated point:

```
>>> points = np.array([[0, 0], [0, 1], [1, 0], [1, 1], [1, 1]])
>>> tri = Delaunay(points)
>>> np.unique(tri.simplices.ravel())
array([0, 1, 2, 3], dtype=int32)
```

Observe that point #4, which is a duplicate, does not occur as a vertex of the triangulation. That this happened is recorded:

```
>>> tri.coplanar
array([[4, 0, 3]], dtype=int32)
```

This means that point 4 resides near triangle 0 and vertex 3, but is not included in the triangulation.

Note that such degeneracies can occur not only because of duplicated points, but also for more complicated geometrical reasons, even in point sets that at first sight seem well-behaved.

However, Qhull has the "QJ" option, which instructs it to perturb the input data randomly until degeneracies are resolved:

```
>>> tri = Delaunay(points, qhull_options="QJ Pp")
>>> points[tri.simplices]
array([[[1, 0],
        [1, 1],
        [0, 0]],
       [[1, 1],
        [1, 1],
        [1, 0]],
       [[1, 1],
        [0, 1],
        [0, 0]],
       [[0, 1],
        [1, 1],
        [1, 1]]])
```

Two new triangles appeared. However, we see that they are degenerate and have zero area.

Convex hulls

Convex hull is the smallest convex object containing all points in a given point set.

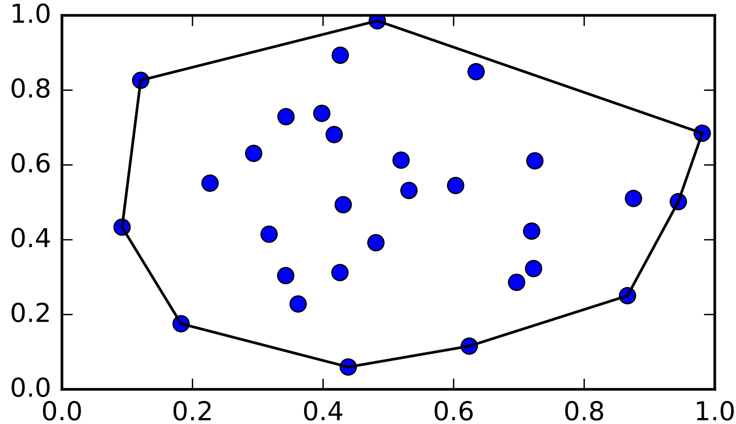
These can be computed via the Qhull wrappers in *scipy.spatial* as follows:

```
>>> from scipy.spatial import ConvexHull
>>> points = np.random.rand(30, 2)   # 30 random points in 2-D
>>> hull = ConvexHull(points)
```

The convex hull is represented as a set of N-1 dimensional simplices, which in 2-D means line segments. The storage scheme is exactly the same as for the simplices in the Delaunay triangulation discussed above.

We can illustrate the above result:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(points[:,0], points[:,1], 'o')
>>> for simplex in hull.simplices:
...     plt.plot(points[simplex,0], points[simplex,1], 'k-')
>>> plt.show()
```



The same can be achieved with *scipy.spatial.convex_hull_plot_2d*.

Voronoi diagrams

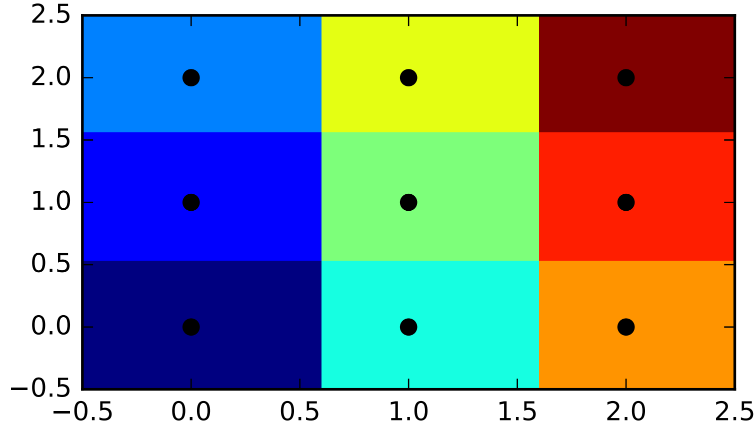
A Voronoi diagram is a subdivision of the space into the nearest neighborhoods of a given set of points.

There are two ways to approach this object using *scipy.spatial*. First, one can use the *KDTree* to answer the question "which of the points is closest to this one", and define the regions that way:

```
>>> from scipy.spatial import KDTree
>>> points = np.array([[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2],
...                    [2, 0], [2, 1], [2, 2]])
>>> tree = KDTree(points)
>>> tree.query([0.1, 0.1])
(0.14142135623730953, 0)
```

So the point `(0.1, 0.1)` belongs to region `0`. In color:

```
>>> x = np.linspace(-0.5, 2.5, 31)
>>> y = np.linspace(-0.5, 2.5, 33)
>>> xx, yy = np.meshgrid(x, y)
>>> xy = np.c_[xx.ravel(), yy.ravel()]
>>> import matplotlib.pyplot as plt
>>> plt.pcolor(x, y, tree.query(xy)[1].reshape(33, 31))
>>> plt.plot(points[:,0], points[:,1], 'ko')
>>> plt.show()
```



This does not, however, give the Voronoi diagram as a geometrical object.

The representation in terms of lines and points can be again obtained via the Qhull wrappers in *scipy.spatial*:

```
>>> from scipy.spatial import Voronoi
>>> vor = Voronoi(points)
>>> vor.vertices
array([[ 0.5,  0.5],
       [ 1.5,  0.5],
       [ 0.5,  1.5],
       [ 1.5,  1.5]])
```

The Voronoi vertices denote the set of points forming the polygonal edges of the Voronoi regions. In this case, there are 9 different regions:

```
>>> vor.regions
[[], [-1, 0], [-1, 1], [1, -1, 0], [3, -1, 2], [-1, 3], [-1, 2], [3, 2, 0, 1], [2, -1,
↪ 0], [3, -1, 1]]
```

Negative value `-1` again indicates a point at infinity. Indeed, only one of the regions, `[3, 1, 0, 2]`, is bounded. Note here that due to similar numerical precision issues as in Delaunay triangulation above, there may be fewer Voronoi regions than input points.

The ridges (lines in 2-D) separating the regions are described as a similar collection of simplices as the convex hull pieces:

```
>>> vor.ridge_vertices
[[-1, 0], [-1, 0], [-1, 1], [-1, 1], [0, 1], [-1, 3], [-1, 2], [2, 3], [-1, 3], [-1,␣
↪2], [0, 2], [1, 3]]
```

These numbers indicate indices of the Voronoi vertices making up the line segments. `-1` is again a point at infinity — only four of the 12 lines is a bounded line segment while the others extend to infinity.

The Voronoi ridges are perpendicular to lines drawn between the input points. Which two points each ridge corresponds to is also recorded:

```
>>> vor.ridge_points
array([[0, 1],
       [0, 3],
```

```
       [6, 3],
       [6, 7],
       [3, 4],
       [5, 8],
       [5, 2],
       [5, 4],
       [8, 7],
       [2, 1],
       [4, 1],
       [4, 7]], dtype=int32)
```

This information, taken together, is enough to construct the full diagram.

We can plot it as follows. First the points and the Voronoi vertices:

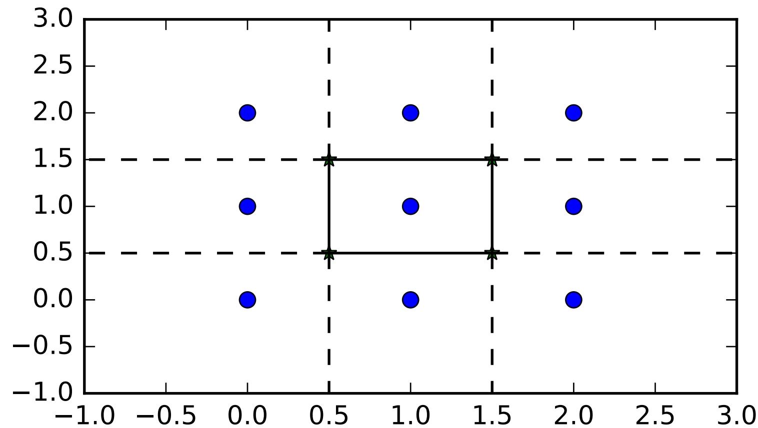
```
>>> plt.plot(points[:, 0], points[:, 1], 'o')
>>> plt.plot(vor.vertices[:, 0], vor.vertices[:, 1], '*')
>>> plt.xlim(-1, 3); plt.ylim(-1, 3)
```

Plotting the finite line segments goes as for the convex hull, but now we have to guard for the infinite edges:

```
>>> for simplex in vor.ridge_vertices:
...     simplex = np.asarray(simplex)
...     if np.all(simplex >= 0):
...         plt.plot(vor.vertices[simplex, 0], vor.vertices[simplex, 1], 'k-')
```

The ridges extending to infinity require a bit more care:

```
>>> center = points.mean(axis=0)
>>> for pointidx, simplex in zip(vor.ridge_points, vor.ridge_vertices):
...     simplex = np.asarray(simplex)
...     if np.any(simplex < 0):
...         i = simplex[simplex >= 0][0] # finite end Voronoi vertex
...         t = points[pointidx[1]] - points[pointidx[0]]  # tangent
...         t = t / np.linalg.norm(t)
...         n = np.array([-t[1], t[0]]) # normal
...         midpoint = points[pointidx].mean(axis=0)
...         far_point = vor.vertices[i] + np.sign(np.dot(midpoint - center, n)) * n *␣
↪100
...         plt.plot([vor.vertices[i,0], far_point[0]],
...                  [vor.vertices[i,1], far_point[1]], 'k--')
>>> plt.show()
```



This plot can also be created using `scipy.spatial.voronoi_plot_2d`.

3.1.13 Statistics (`scipy.stats`)

Introduction

In this tutorial we discuss many, but certainly not all, features of `scipy.stats`. The intention here is to provide a user with a working knowledge of this package. We refer to the reference manual for further details.

Note: This documentation is work in progress.

Random Variables

There are two general distribution classes that have been implemented for encapsulating continuous random variables and discrete random variables . Over 80 continuous random variables (RVs) and 10 discrete random variables have been implemented using these classes. Besides this, new routines and distributions can easily added by the end user. (If you create one, please contribute it).

All of the statistics functions are located in the sub-package `scipy.stats` and a fairly complete listing of these functions can be obtained using `info(stats)`. The list of the random variables available can also be obtained from the docstring for the stats sub-package.

In the discussion below we mostly focus on continuous RVs. Nearly all applies to discrete variables also, but we point out some differences here: *Specific Points for Discrete Distributions*.

In the code samples below we assume that the `scipy.stats` package is imported as

```
>>> from scipy import stats
```

and in some cases we assume that individual objects are imported as

```
>>> from scipy.stats import norm
```

Getting Help

First of all, all distributions are accompanied with help functions. To obtain just some basic information we print the relevant docstring: `print(stats.norm.__doc__)`.

To find the support, i.e., upper and lower bound of the distribution, call:

```
>>> print 'bounds of distribution lower: %s, upper: %s' % (norm.a, norm.b)
bounds of distribution lower: -inf, upper: inf
```

We can list all methods and properties of the distribution with `dir(norm)`. As it turns out, some of the methods are private methods although they are not named as such (their name does not start with a leading underscore), for example `veccdf`, are only available for internal calculation (those methods will give warnings when one tries to use them, and will be removed at some point).

To obtain the *real* main methods, we list the methods of the frozen distribution. (We explain the meaning of a *frozen* distribution below).

```
>>> rv = norm()
>>> dir(rv)  # reformatted
    ['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
    '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__',
    '__repr__', '__setattr__', '__str__', '__weakref__', 'args', 'cdf', 'dist',
    'entropy', 'isf', 'kwds', 'moment', 'pdf', 'pmf', 'ppf', 'rvs', 'sf', 'stats']
```

Finally, we can obtain the list of available distribution through introspection:

```
>>> import warnings
>>> warnings.simplefilter('ignore', DeprecationWarning)
>>> dist_continu = [d for d in dir(stats) if
...                 isinstance(getattr(stats,d), stats.rv_continuous)]
>>> dist_discrete = [d for d in dir(stats) if
...                  isinstance(getattr(stats,d), stats.rv_discrete)]
>>> print 'number of continuous distributions:', len(dist_continu)
number of continuous distributions: 95
>>> print 'number of discrete distributions:  ', len(dist_discrete)
number of discrete distributions:   13
```

Common Methods

The main public methods for continuous RVs are:

- rvs: Random Variates
- pdf: Probability Density Function
- cdf: Cumulative Distribution Function
- sf: Survival Function (1-CDF)
- ppf: Percent Point Function (Inverse of CDF)
- isf: Inverse Survival Function (Inverse of SF)
- stats: Return mean, variance, (Fisher's) skew, or (Fisher's) kurtosis
- moment: non-central moments of the distribution

Let's take a normal RV as an example.

```
>>> norm.cdf(0)
0.5
```

To compute the `cdf` at a number of points, we can pass a list or a numpy array.

```
>>> norm.cdf([-1., 0, 1])
array([ 0.15865525,  0.5,  0.84134475])
>>> import numpy as np
>>> norm.cdf(np.array([-1., 0, 1]))
array([ 0.15865525,  0.5,  0.84134475])
```

Thus, the basic methods such as *pdf*, *cdf*, and so on are vectorized with `np.vectorize`.

Other generally useful methods are supported too:

```
>>> norm.mean(), norm.std(), norm.var()
(0.0, 1.0, 1.0)
>>> norm.stats(moments = "mv")
(array(0.0), array(1.0))
```

To find the median of a distribution we can use the percent point function `ppf`, which is the inverse of the `cdf`:

```
>>> norm.ppf(0.5)
0.0
```

To generate a sequence of random variates, use the `size` keyword argument:

```
>>> norm.rvs(size=3)
array([-0.35687759,  1.34347647, -0.11710531])   # random
```

Note that drawing random numbers relies on generators from `numpy.random` package. In the example above, the specific stream of random numbers is not reproducible across runs. To achieve reproducibility, you can explicitly seed a global variable

```
>>> np.random.seed(1234)
```

Relying on a global state is not recommended though. A better way is to use the *random_state* parameter which accepts an instance of `numpy.random.RandomState` class, or an integer which is then used to seed an internal *RandomState* object:

```
>>> norm.rvs(size=5, random_state=1234)
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873])
```

Don't think that `norm.rvs(5)` generates 5 variates:

```
>>> norm.rvs(5)
5.471435163732493
```

Here, `5` with no keyword is being interpreted as the first possible keyword argument, `loc`, which is the first of a pair of keyword arguments taken by all continuous distributions. This brings us to the topic of the next subsection.

Shifting and Scaling

All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution, e.g. for the standard normal distribution the location is the mean and the scale is the standard deviation.

```
>>> norm.stats(loc = 3, scale = 4, moments = "mv")
(array(3.0), array(16.0))
```

In many cases the standardized distribution for a random variable `X` is obtained through the transformation `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

Smart use of `loc` and `scale` can help modify the standard distributions in many ways. To illustrate the scaling further, the `cdf` of an exponentially distributed RV with mean $1/\lambda$ is given by

$$F(x) = 1 - \exp(-\lambda x)$$

By applying the scaling rule above, it can be seen that by taking `scale = 1./lambda` we get the proper scale.

```
>>> from scipy.stats import expon
>>> expon.mean(scale=3.)
3.0
```

Note: Distributions that take shape parameters may require more than simple application of `loc` and/or `scale` to achieve the desired form. For example, the distribution of 2-D vector lengths given a constant vector of length R perturbed by independent N(0, σ^2) deviations in each component is rice(R/σ, scale= σ). The first argument is a shape parameter that needs to be scaled along with x.

The uniform distribution is also interesting:

```
>>> from scipy.stats import uniform
>>> uniform.cdf([0, 1, 2, 3, 4, 5], loc = 1, scale = 4)
array([ 0.  ,  0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

Finally, recall from the previous paragraph that we are left with the problem of the meaning of `norm.rvs(5)`. As it turns out, calling a distribution like this, the first argument, i.e., the 5, gets passed to set the `loc` parameter. Let's see:

```
>>> np.mean(norm.rvs(5, size=500))
4.983550784784704
```

Thus, to explain the output of the example of the last section: `norm.rvs(5)` generates a single normally distributed random variate with mean `loc=5`, because of the default `size=1`.

We recommend that you set `loc` and `scale` parameters explicitly, by passing the values as keywords rather than as arguments. Repetition can be minimized when calling more than one method of a given RV by using the technique of *Freezing a Distribution*, as explained below.

Shape Parameters

While a general continuous random variable can be shifted and scaled with the `loc` and `scale` parameters, some distributions require additional shape parameters. For instance, the gamma distribution, with density

$$\gamma(x, a) = \frac{\lambda(\lambda x)^{a-1}}{\Gamma(a)} e^{-\lambda x} \,,$$

requires the shape parameter a. Observe that setting λ can be obtained by setting the `scale` keyword to $1/\lambda$.

Let's check the number and name of the shape parameters of the gamma distribution. (We know from the above that this should be 1.)

```
>>> from scipy.stats import gamma
>>> gamma.numargs
1
>>> gamma.shapes
'a'
```

Now we set the value of the shape variable to 1 to obtain the exponential distribution, so that we compare easily whether we get the results we expect.

```
>>> gamma(1, scale=2.).stats(moments="mv")
(array(2.0), array(4.0))
```

Notice that we can also specify shape parameters as keywords:

```
>>> gamma(a=1, scale=2.).stats(moments="mv")
(array(2.0), array(4.0))
```

Freezing a Distribution

Passing the `loc` and `scale` keywords time and again can become quite bothersome. The concept of *freezing* a RV is used to solve such problems.

```
>>> rv = gamma(1, scale=2.)
```

By using `rv` we no longer have to include the scale or the shape parameters anymore. Thus, distributions can be used in one of two ways, either by passing all distribution parameters to each method call (such as we did earlier) or by freezing the parameters for the instance of the distribution. Let us check this:

```
>>> rv.mean(), rv.std()
(2.0, 2.0)
```

This is indeed what we should get.

Broadcasting

The basic methods `pdf` and so on satisfy the usual numpy broadcasting rules. For example, we can calculate the critical values for the upper tail of the t distribution for different probabilities and degrees of freedom.

```
>>> stats.t.isf([0.1, 0.05, 0.01], [[10], [11]])
array([[ 1.37218364,  1.81246112,  2.76376946],
       [ 1.36343032,  1.79588482,  2.71807918]])
```

Here, the first row are the critical values for 10 degrees of freedom and the second row for 11 degrees of freedom (d.o.f.). Thus, the broadcasting rules give the same result of calling `isf` twice:

```
>>> stats.t.isf([0.1, 0.05, 0.01], 10)
array([ 1.37218364,  1.81246112,  2.76376946])
>>> stats.t.isf([0.1, 0.05, 0.01], 11)
array([ 1.36343032,  1.79588482,  2.71807918])
```

If the array with probabilities, i.e., `[0.1, 0.05, 0.01]` and the array of degrees of freedom i.e., `[10, 11, 12]`, have the same array shape, then element wise matching is used. As an example, we can obtain the 10% tail for 10 d.o.f., the 5% tail for 11 d.o.f. and the 1% tail for 12 d.o.f. by calling

```
>>> stats.t.isf([0.1, 0.05, 0.01], [10, 11, 12])
array([ 1.37218364,  1.79588482,  2.68099799])
```

Specific Points for Discrete Distributions

Discrete distribution have mostly the same basic methods as the continuous distributions. However `pdf` is replaced the probability mass function `pmf`, no estimation methods, such as fit, are available, and `scale` is not a valid keyword parameter. The location parameter, keyword `loc` can still be used to shift the distribution.

The computation of the cdf requires some extra attention. In the case of continuous distribution the cumulative distribution function is in most standard cases strictly monotonic increasing in the bounds (a,b) and has therefore a unique

inverse. The cdf of a discrete distribution, however, is a step function, hence the inverse cdf, i.e., the percent point function, requires a different definition:

```
ppf(q) = min{x : cdf(x) >= q, x integer}
```

For further info, see the docs here.

We can look at the hypergeometric distribution as an example

```
>>> from scipy.stats import hypergeom
>>> [M, n, N] = [20, 7, 12]
```

If we use the cdf at some integer points and then evaluate the ppf at those cdf values, we get the initial integers back, for example

```
>>> x = np.arange(4)*2
>>> x
array([0, 2, 4, 6])
>>> prb = hypergeom.cdf(x, M, n, N)
>>> prb
array([ 0.0001031991744066,  0.0521155830753351,  0.6083591331269301,
        0.9897832817337386])
>>> hypergeom.ppf(prb, M, n, N)
array([ 0.,  2.,  4.,  6.])
```

If we use values that are not at the kinks of the cdf step function, we get the next higher integer back:

```
>>> hypergeom.ppf(prb + 1e-8, M, n, N)
array([ 1.,  3.,  5.,  7.])
>>> hypergeom.ppf(prb - 1e-8, M, n, N)
array([ 0.,  2.,  4.,  6.])
```

Fitting Distributions

The main additional methods of the not frozen distribution are related to the estimation of distribution parameters:

- ***fit: maximum likelihood estimation of distribution parameters, including location***
  and scale
- fit_loc_scale: estimation of location and scale when shape parameters are given
- nnlf: negative log likelihood function
- expect: Calculate the expectation of a function against the pdf or pmf

Performance Issues and Cautionary Remarks

The performance of the individual methods, in terms of speed, varies widely by distribution and method. The results of a method are obtained in one of two ways: either by explicit calculation, or by a generic algorithm that is independent of the specific distribution.

Explicit calculation, on the one hand, requires that the method is directly specified for the given distribution, either through analytic formulas or through special functions in `scipy.special` or `numpy.random` for `rvs`. These are usually relatively fast calculations.

The generic methods, on the other hand, are used if the distribution does not specify any explicit calculation. To define a distribution, only one of pdf or cdf is necessary; all other methods can be derived using numeric integration and root finding. However, these indirect methods can be *very* slow. As an example, `rgh = stats.gausshyper.rvs(0.5, 2, 2, 2, size=100)` creates random variables in a very indirect way and takes about 19 seconds for 100 random variables on my computer, while one million random variables from the standard normal or from the t distribution take just above one second.

Remaining Issues

The distributions in `scipy.stats` have recently been corrected and improved and gained a considerable test suite, however a few issues remain:

- the distributions have been tested over some range of parameters, however in some corner ranges, a few incorrect results may remain.
- the maximum likelihood estimation in *fit* does not work with default starting parameters for all distributions and the user needs to supply good starting parameters. Also, for some distribution using a maximum likelihood estimator might inherently not be the best choice.

Building Specific Distributions

The next examples shows how to build your own distributions. Further examples show the usage of the distributions and some statistical tests.

Making a Continuous Distribution, i.e., Subclassing `rv_continuous`

Making continuous distributions is fairly simple.

```
>>> from scipy import stats
>>> class deterministic_gen(stats.rv_continuous):
...     def _cdf(self, x):
...         return np.where(x < 0, 0., 1.)
...     def _stats(self):
...         return 0., 0., 0., 0.
```

```
>>> deterministic = deterministic_gen(name="deterministic")
>>> deterministic.cdf(np.arange(-3, 3, 0.5))
array([ 0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Interestingly, the `pdf` is now computed automatically:

```
>>> deterministic.pdf(np.arange(-3, 3, 0.5))
array([  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
         5.83333333e+04,   4.16333634e-12,   4.16333634e-12,
         4.16333634e-12,   4.16333634e-12,   4.16333634e-12])
```

Be aware of the performance issues mentions in *Performance Issues and Cautionary Remarks*. The computation of unspecified common methods can become very slow, since only general methods are called which, by their very nature, cannot use any specific information about the distribution. Thus, as a cautionary example:

```
>>> from scipy.integrate import quad
>>> quad(deterministic.pdf, -1e-1, 1e-1)
(4.163336342344337e-13, 0.0)
```

But this is not correct: the integral over this pdf should be 1. Let's make the integration interval smaller:

```
>>> quad(deterministic.pdf, -1e-3, 1e-3)  # warning removed
(1.000076872229173, 0.0010625571718182458)
```

This looks better. However, the problem originated from the fact that the pdf is not specified in the class definition of the deterministic distribution.

Subclassing `rv_discrete`

In the following we use `stats.rv_discrete` to generate a discrete distribution that has the probabilities of the truncated normal for the intervals centered around the integers.

General Info

From the docstring of rv_discrete, `help(stats.rv_discrete)`,

> "You can construct an arbitrary discrete rv where P{X=xk} = pk by passing to the rv_discrete initialization method (through the values= keyword) a tuple of sequences (xk, pk) which describes only those values of X (xk) that occur with nonzero probability (pk)."

Next to this, there are some further requirements for this approach to work:

- The keyword *name* is required.
- The support points of the distribution xk have to be integers.
- The number of significant digits (decimals) needs to be specified.

In fact, if the last two requirements are not satisfied an exception may be raised or the resulting numbers may be incorrect.

An Example

Let's do the work. First

```
>>> npoints = 20   # number of integer support points of the distribution minus 1
>>> npointsh = npoints / 2
>>> npointsf = float(npoints)
>>> nbound = 4   # bounds for the truncated normal
>>> normbound = (1+1/npointsf) * nbound   # actual bounds of truncated normal
>>> grid = np.arange(-npointsh, npointsh+2, 1)   # integer grid
>>> gridlimitsnorm = (grid-0.5) / npointsh * nbound   # bin limits for the truncnorm
>>> gridlimits = grid - 0.5   # used later in the analysis
>>> grid = grid[:-1]
>>> probs = np.diff(stats.truncnorm.cdf(gridlimitsnorm, -normbound, normbound))
>>> gridint = grid
```

And finally we can subclass `rv_discrete`:

```
>>> normdiscrete = stats.rv_discrete(values=(gridint,
...              np.round(probs, decimals=7)), name='normdiscrete')
```

Now that we have defined the distribution, we have access to all common methods of discrete distributions.

```
>>> print 'mean = %6.4f, variance = %6.4f, skew = %6.4f, kurtosis = %6.4f'% \
...       normdiscrete.stats(moments =  'mvsk')
mean = -0.0000, variance = 6.3302, skew = 0.0000, kurtosis = -0.0076
```

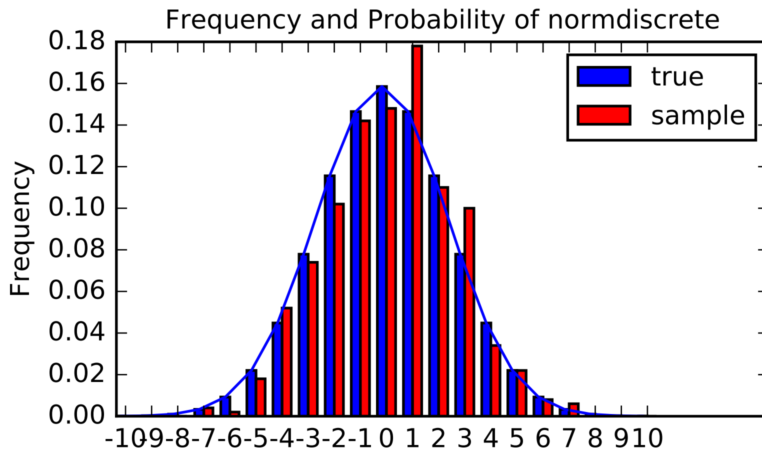
```
>>> nd_std = np.sqrt(normdiscrete.stats(moments='v'))
```

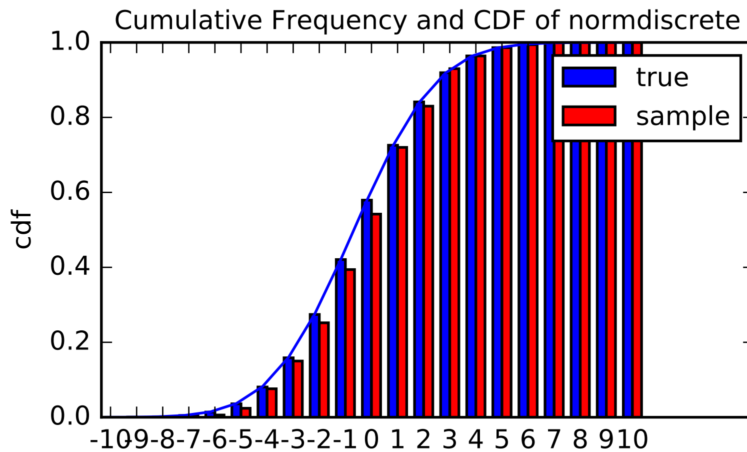
Testing the Implementation

Let's generate a random sample and compare observed frequencies with the probabilities.

```
>>> n_sample = 500
>>> np.random.seed(87655678)   # fix the seed for replicability
>>> rvs = normdiscrete.rvs(size=n_sample)
>>> rvsnd = rvs
>>> f, l = np.histogram(rvs, bins=gridlimits)
```

```
>>> sfreq = np.vstack([gridint, f, probs*n_sample]).T
>>> print sfreq
[[ -1.00000000e+01   0.00000000e+00   2.95019349e-02]
 [ -9.00000000e+00   0.00000000e+00   1.32294142e-01]
 [ -8.00000000e+00   0.00000000e+00   5.06497902e-01]
 [ -7.00000000e+00   2.00000000e+00   1.65568919e+00]
 [ -6.00000000e+00   1.00000000e+00   4.62125309e+00]
 [ -5.00000000e+00   9.00000000e+00   1.10137298e+01]
 [ -4.00000000e+00   2.60000000e+01   2.24137683e+01]
 [ -3.00000000e+00   3.70000000e+01   3.89503370e+01]
 [ -2.00000000e+00   5.10000000e+01   5.78004747e+01]
 [ -1.00000000e+00   7.10000000e+01   7.32455414e+01]
 [  0.00000000e+00   7.40000000e+01   7.92618251e+01]
 [  1.00000000e+00   8.90000000e+01   7.32455414e+01]
 [  2.00000000e+00   5.50000000e+01   5.78004747e+01]
 [  3.00000000e+00   5.00000000e+01   3.89503370e+01]
 [  4.00000000e+00   1.70000000e+01   2.24137683e+01]
 [  5.00000000e+00   1.10000000e+01   1.10137298e+01]
 [  6.00000000e+00   4.00000000e+00   4.62125309e+00]
 [  7.00000000e+00   3.00000000e+00   1.65568919e+00]
 [  8.00000000e+00   0.00000000e+00   5.06497902e-01]
 [  9.00000000e+00   0.00000000e+00   1.32294142e-01]
 [  1.00000000e+01   0.00000000e+00   2.95019349e-02]]
```





Next, we can test, whether our sample was generated by our normdiscrete distribution. This also verifies whether the random numbers are generated correctly.

The chisquare test requires that there are a minimum number of observations in each bin. We combine the tail bins into larger bins so that they contain enough observations.

```
>>> f2 = np.hstack([f[:5].sum(), f[5:-5], f[-5:].sum()])
>>> p2 = np.hstack([probs[:5].sum(), probs[5:-5], probs[-5:].sum()])
>>> ch2, pval = stats.chisquare(f2, p2*n_sample)
```

```
>>> print 'chisquare for normdiscrete: chi2 = %6.3f pvalue = %6.4f' % (ch2, pval)
chisquare for normdiscrete: chi2 = 12.466 pvalue = 0.4090
```

The pvalue in this case is high, so we can be quite confident that our random sample was actually generated by the distribution.

Analysing One Sample

First, we create some random variables. We set a seed so that in each run we get identical results to look at. As an example we take a sample from the Student t distribution:

```
>>> np.random.seed(282629734)
>>> x = stats.t.rvs(10, size=1000)
```

Here, we set the required shape parameter of the t distribution, which in statistics corresponds to the degrees of freedom, to 10. Using size=1000 means that our sample consists of 1000 independently drawn (pseudo) random numbers. Since we did not specify the keyword arguments *loc* and *scale*, those are set to their default values zero and one.

Descriptive Statistics

x is a numpy array, and we have direct access to all array methods, e.g.

```
>>> print x.max(), x.min()  # equivalent to np.max(x), np.min(x)
5.26327732981 -3.78975572422
>>> print x.mean(), x.var() # equivalent to np.mean(x), np.var(x)
0.0140610663985 1.28899386208
```

How do the some sample properties compare to their theoretical counterparts?

```
>>> m, v, s, k = stats.t.stats(10, moments='mvsk')
>>> n, (smin, smax), sm, sv, ss, sk = stats.describe(x)
```

```
>>> print 'distribution:',
distribution:
>>> sstr = 'mean = %6.4f, variance = %6.4f, skew = %6.4f, kurtosis = %6.4f'
>>> print sstr %(m, v, s ,k)
mean = 0.0000, variance = 1.2500, skew = 0.0000, kurtosis = 1.0000
>>> print 'sample:      ',
sample:
>>> print sstr %(sm, sv, ss, sk)
mean = 0.0141, variance = 1.2903, skew = 0.2165, kurtosis = 1.0556
```

Note: stats.describe uses the unbiased estimator for the variance, while np.var is the biased estimator.

For our sample the sample statistics differ a by a small amount from their theoretical counterparts.

T-test and KS-test

We can use the t-test to test whether the mean of our sample differs in a statistically significant way from the theoretical expectation.

```
>>> print 't-statistic = %6.3f pvalue = %6.4f' %  stats.ttest_1samp(x, m)
t-statistic =  0.391 pvalue = 0.6955
```

The pvalue is 0.7, this means that with an alpha error of, for example, 10%, we cannot reject the hypothesis that the sample mean is equal to zero, the expectation of the standard t-distribution.

As an exercise, we can calculate our ttest also directly without using the provided function, which should give us the same answer, and so it does:

```
>>> tt = (sm-m)/np.sqrt(sv/float(n))  # t-statistic for mean
>>> pval = stats.t.sf(np.abs(tt), n-1)*2  # two-sided pvalue = Prob(abs(t)>tt)
>>> print 't-statistic = %6.3f pvalue = %6.4f' % (tt, pval)
t-statistic =  0.391 pvalue = 0.6955
```

The Kolmogorov-Smirnov test can be used to test the hypothesis that the sample comes from the standard t-distribution

```
>>> print 'KS-statistic D = %6.3f pvalue = %6.4f' % stats.kstest(x, 't', (10,))
KS-statistic D =  0.016 pvalue = 0.9606
```

Again the p-value is high enough that we cannot reject the hypothesis that the random sample really is distributed according to the t-distribution. In real applications, we don't know what the underlying distribution is. If we perform the Kolmogorov-Smirnov test of our sample against the standard normal distribution, then we also cannot reject the hypothesis that our sample was generated by the normal distribution given that in this example the p-value is almost 40%.

```
>>> print 'KS-statistic D = %6.3f pvalue = %6.4f' % stats.kstest(x,'norm')
KS-statistic D =  0.028 pvalue = 0.3949
```

However, the standard normal distribution has a variance of 1, while our sample has a variance of 1.29. If we standardize our sample and test it against the normal distribution, then the p-value is again large enough that we cannot reject the hypothesis that the sample came form the normal distribution.

```
>>> d, pval = stats.kstest((x-x.mean())/x.std(), 'norm')
>>> print 'KS-statistic D = %6.3f pvalue = %6.4f' % (d, pval)
KS-statistic D =  0.032 pvalue = 0.2402
```

Note: The Kolmogorov-Smirnov test assumes that we test against a distribution with given parameters, since in the last case we estimated mean and variance, this assumption is violated, and the distribution of the test statistic on which the p-value is based, is not correct.

Tails of the distribution

Finally, we can check the upper tail of the distribution. We can use the percent point function ppf, which is the inverse of the cdf function, to obtain the critical values, or, more directly, we can use the inverse of the survival function

```
>>> crit01, crit05, crit10 = stats.t.ppf([1-0.01, 1-0.05, 1-0.10], 10)
>>> print 'critical values from ppf at 1%%, 5%% and 10%% %8.4f %8.4f %8.4f'% (crit01,
↪crit05, crit10)
critical values from ppf at 1%, 5% and 10%   2.7638   1.8125   1.3722
>>> print 'critical values from isf at 1%%, 5%% and 10%% %8.4f %8.4f %8.4f'%
↪tuple(stats.t.isf([0.01,0.05,0.10],10))
critical values from isf at 1%, 5% and 10%   2.7638   1.8125   1.3722
```

```
>>> freq01 = np.sum(x>crit01) / float(n) * 100
>>> freq05 = np.sum(x>crit05) / float(n) * 100
>>> freq10 = np.sum(x>crit10) / float(n) * 100
>>> print 'sample %%-frequency at 1%%, 5%% and 10%% tail %8.4f %8.4f %8.4f'% (freq01,
↪freq05, freq10)
sample %-frequency at 1%, 5% and 10% tail   1.4000   5.8000  10.5000
```

In all three cases, our sample has more weight in the top tail than the underlying distribution. We can briefly check a larger sample to see if we get a closer match. In this case the empirical frequency is quite close to the theoretical probability, but if we repeat this several times the fluctuations are still pretty large.

```
>>> freq05l = np.sum(stats.t.rvs(10, size=10000) > crit05) / 10000.0 * 100
>>> print 'larger sample %%-frequency at 5%% tail %8.4f'% freq05l
larger sample %-frequency at 5% tail   4.8000
```

We can also compare it with the tail of the normal distribution, which has less weight in the tails:

```
>>> print 'tail prob. of normal at 1%%, 5%% and 10%% %8.4f %8.4f %8.4f'% \
...       tuple(stats.norm.sf([crit01, crit05, crit10])*100)
tail prob. of normal at 1%, 5% and 10%   0.2857   3.4957   8.5003
```

The chisquare test can be used to test, whether for a finite number of bins, the observed frequencies differ significantly from the probabilities of the hypothesized distribution.

```
>>> quantiles = [0.0, 0.01, 0.05, 0.1, 1-0.10, 1-0.05, 1-0.01, 1.0]
>>> crit = stats.t.ppf(quantiles, 10)
>>> crit
array([-Inf, -2.76376946, -1.81246112, -1.37218364, 1.37218364, 1.81246112,
  2.76376946, Inf])
>>> n_sample = x.size
>>> freqcount = np.histogram(x, bins=crit)[0]
>>> tprob = np.diff(quantiles)
>>> nprob = np.diff(stats.norm.cdf(crit))
>>> tch, tpval = stats.chisquare(freqcount, tprob*n_sample)
>>> nch, npval = stats.chisquare(freqcount, nprob*n_sample)
>>> print 'chisquare for t:      chi2 = %6.2f pvalue = %6.4f' % (tch, tpval)
chisquare for t:      chi2 =  2.30 pvalue = 0.8901
>>> print 'chisquare for normal: chi2 = %6.2f pvalue = %6.4f' % (nch, npval)
chisquare for normal: chi2 = 64.60 pvalue = 0.0000
```

We see that the standard normal distribution is clearly rejected while the standard t-distribution cannot be rejected. Since the variance of our sample differs from both standard distribution, we can again redo the test taking the estimate for scale and location into account.

The fit method of the distributions can be used to estimate the parameters of the distribution, and the test is repeated using probabilities of the estimated distribution.

```
>>> tdof, tloc, tscale = stats.t.fit(x)
>>> nloc, nscale = stats.norm.fit(x)
>>> tprob = np.diff(stats.t.cdf(crit, tdof, loc=tloc, scale=tscale))
>>> nprob = np.diff(stats.norm.cdf(crit, loc=nloc, scale=nscale))
>>> tch, tpval = stats.chisquare(freqcount, tprob*n_sample)
>>> nch, npval = stats.chisquare(freqcount, nprob*n_sample)
>>> print 'chisquare for t:      chi2 = %6.2f pvalue = %6.4f' % (tch, tpval)
chisquare for t:      chi2 =  1.58 pvalue = 0.9542
>>> print 'chisquare for normal: chi2 = %6.2f pvalue = %6.4f' % (nch, npval)
chisquare for normal: chi2 = 11.08 pvalue = 0.0858
```

Taking account of the estimated parameters, we can still reject the hypothesis that our sample came from a normal distribution (at the 5% level), but again, with a p-value of 0.95, we cannot reject the t distribution.

Special tests for normal distributions

Since the normal distribution is the most common distribution in statistics, there are several additional functions available to test whether a sample could have been drawn from a normal distribution

First we can test if skew and kurtosis of our sample differ significantly from those of a normal distribution:

```
>>> print 'normal skewtest teststat = %6.3f pvalue = %6.4f' % stats.skewtest(x)
normal skewtest teststat =  2.785 pvalue = 0.0054
>>> print 'normal kurtosistest teststat = %6.3f pvalue = %6.4f' % stats.
↪kurtosistest(x)
normal kurtosistest teststat =  4.757 pvalue = 0.0000
```

These two tests are combined in the normality test

```
>>> print 'normaltest teststat = %6.3f pvalue = %6.4f' % stats.normaltest(x)
normaltest teststat = 30.379 pvalue = 0.0000
```

In all three tests the p-values are very low and we can reject the hypothesis that the our sample has skew and kurtosis of the normal distribution.

Since skew and kurtosis of our sample are based on central moments, we get exactly the same results if we test the standardized sample:

```
>>> print 'normaltest teststat = %6.3f pvalue = %6.4f' % \
...                       stats.normaltest((x-x.mean())/x.std())
normaltest teststat = 30.379 pvalue = 0.0000
```

Because normality is rejected so strongly, we can check whether the normaltest gives reasonable results for other cases:

```
>>> print('normaltest teststat = %6.3f pvalue = %6.4f' %
...               stats.normaltest(stats.t.rvs(10, size=100)))
normaltest teststat =  4.698 pvalue = 0.0955
>>> print('normaltest teststat = %6.3f pvalue = %6.4f' %
...               stats.normaltest(stats.norm.rvs(size=1000)))
normaltest teststat =  0.613 pvalue = 0.7361
```

When testing for normality of a small sample of t-distributed observations and a large sample of normal distributed observation, then in neither case can we reject the null hypothesis that the sample comes from a normal distribution. In the first case this is because the test is not powerful enough to distinguish a t and a normally distributed random variable in a small sample.

Comparing two samples

In the following, we are given two samples, which can come either from the same or from different distribution, and we want to test whether these samples have the same statistical properties.

Comparing means

Test with sample with identical means:

```
>>> rvs1 = stats.norm.rvs(loc=5, scale=10, size=500)
>>> rvs2 = stats.norm.rvs(loc=5, scale=10, size=500)
>>> stats.ttest_ind(rvs1, rvs2)
(-0.54890361750888583, 0.5831943748663857)
```

Test with sample with different means:

```
>>> rvs3 = stats.norm.rvs(loc=8, scale=10, size=500)
>>> stats.ttest_ind(rvs1, rvs3)
(-4.5334142901750321, 6.507128186505895e-006)
```

Kolmogorov-Smirnov test for two samples ks_2samp

For the example where both samples are drawn from the same distribution, we cannot reject the null hypothesis since the pvalue is high

```
>>> stats.ks_2samp(rvs1, rvs2)
(0.025999999999999995, 0.99541195173064878)
```

In the second example, with different location, i.e. means, we can reject the null hypothesis since the pvalue is below 1%

```
>>> stats.ks_2samp(rvs1, rvs3)
(0.11399999999999999, 0.0027132103661283141)
```

Kernel Density Estimation

A common task in statistics is to estimate the probability density function (PDF) of a random variable from a set of data samples. This task is called density estimation. The most well-known tool to do this is the histogram. A histogram is a useful tool for visualization (mainly because everyone understands it), but doesn't use the available data very efficiently. Kernel density estimation (KDE) is a more efficient tool for the same task. The `gaussian_kde` estimator can be used to estimate the PDF of univariate as well as multivariate data. It works best if the data is unimodal.

Univariate estimation

We start with a minimal amount of data in order to see how `gaussian_kde` works, and what the different options for bandwidth selection do. The data sampled from the PDF is show as blue dashes at the bottom of the figure (this is called a rug plot):

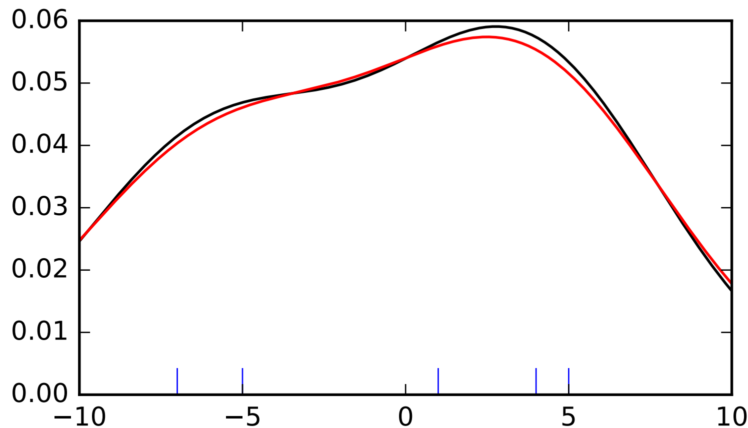
```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

```
>>> x1 = np.array([-7, -5, 1, 4, 5], dtype=np.float)
>>> kde1 = stats.gaussian_kde(x1)
>>> kde2 = stats.gaussian_kde(x1, bw_method='silverman')
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
```

```
>>> ax.plot(x1, np.zeros(x1.shape), 'b+', ms=20)  # rug plot
>>> x_eval = np.linspace(-10, 10, num=200)
>>> ax.plot(x_eval, kde1(x_eval), 'k-', label="Scott's Rule")
>>> ax.plot(x_eval, kde2(x_eval), 'r-', label="Silverman's Rule")
```

```
>>> plt.show()
```



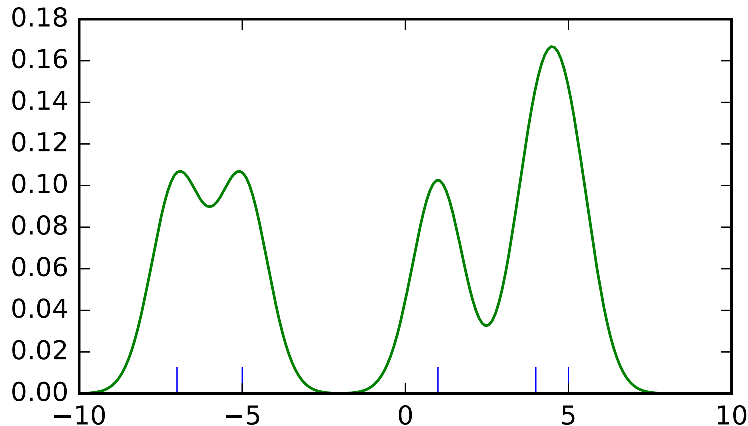
We see that there is very little difference between Scott's Rule and Silverman's Rule, and that the bandwidth selection with a limited amount of data is probably a bit too wide. We can define our own bandwidth function to get a less smoothed out result.

```
>>> def my_kde_bandwidth(obj, fac=1./5):
...     """We use Scott's Rule, multiplied by a constant factor."""
...     return np.power(obj.n, -1./(obj.d+4)) * fac
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
```

```
>>> ax.plot(x1, np.zeros(x1.shape), 'b+', ms=20)  # rug plot
>>> kde3 = stats.gaussian_kde(x1, bw_method=my_kde_bandwidth)
>>> ax.plot(x_eval, kde3(x_eval), 'g-', label="With smaller BW")
```

```
>>> plt.show()
```

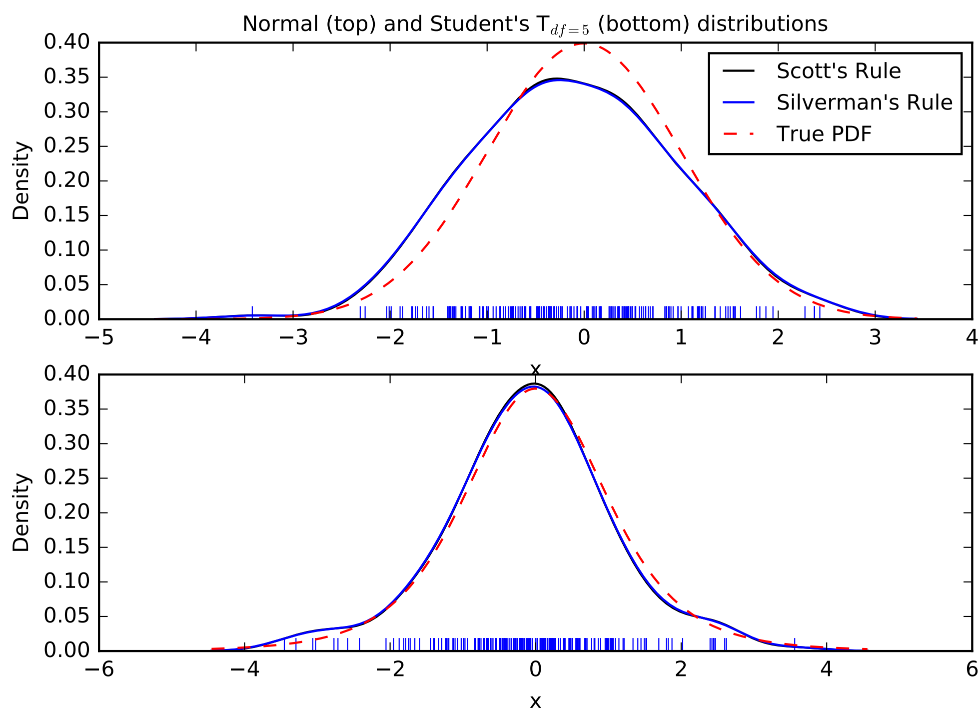


We see that if we set bandwidth to be very narrow, the obtained estimate for the probability density function (PDF) is simply the sum of Gaussians around each data point.

We now take a more realistic example, and look at the difference between the two available bandwidth selection rules. Those rules are known to work well for (close to) normal distributions, but even for unimodal distributions that are quite strongly non-normal they work reasonably well. As a non-normal distribution we take a Student's T distribution with 5 degrees of freedom.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats


np.random.seed(12456)
x1 = np.random.normal(size=200)  # random data, normal distribution
xs = np.linspace(x1.min()-1, x1.max()+1, 200)

kde1 = stats.gaussian_kde(x1)
kde2 = stats.gaussian_kde(x1, bw_method='silverman')

fig = plt.figure(figsize=(8, 6))

ax1 = fig.add_subplot(211)
ax1.plot(x1, np.zeros(x1.shape), 'b+', ms=12)  # rug plot
ax1.plot(xs, kde1(xs), 'k-', label="Scott's Rule")
ax1.plot(xs, kde2(xs), 'b-', label="Silverman's Rule")
ax1.plot(xs, stats.norm.pdf(xs), 'r--', label="True PDF")

ax1.set_xlabel('x')
ax1.set_ylabel('Density')
ax1.set_title("Normal (top) and Student's T$_{df=5}$ (bottom) distributions")
ax1.legend(loc=1)

x2 = stats.t.rvs(5, size=200)  # random data, T distribution
xs = np.linspace(x2.min() - 1, x2.max() + 1, 200)

kde3 = stats.gaussian_kde(x2)
kde4 = stats.gaussian_kde(x2, bw_method='silverman')
```

```
ax2 = fig.add_subplot(212)
ax2.plot(x2, np.zeros(x2.shape), 'b+', ms=12)  # rug plot
ax2.plot(xs, kde3(xs), 'k-', label="Scott's Rule")
ax2.plot(xs, kde4(xs), 'b-', label="Silverman's Rule")
ax2.plot(xs, stats.t.pdf(xs, 5), 'r--', label="True PDF")

ax2.set_xlabel('x')
ax2.set_ylabel('Density')

plt.show()
```



We now take a look at a bimodal distribution with one wider and one narrower Gaussian feature. We expect that this will be a more difficult density to approximate, due to the different bandwidths required to accurately resolve each feature.

```
>>> from functools import partial
```

```
>>> loc1, scale1, size1 = (-2, 1, 175)
>>> loc2, scale2, size2 = (2, 0.2, 50)
>>> x2 = np.concatenate([np.random.normal(loc=loc1, scale=scale1, size=size1),
...                      np.random.normal(loc=loc2, scale=scale2, size=size2)])
```

```
>>> x_eval = np.linspace(x2.min() - 1, x2.max() + 1, 500)
```

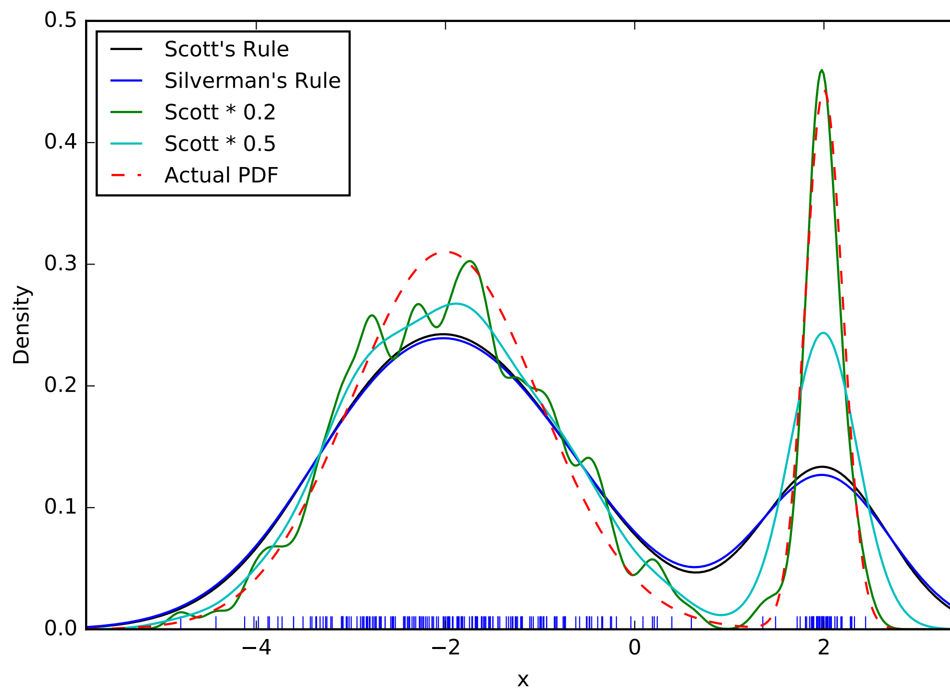
```
>>> kde = stats.gaussian_kde(x2)
>>> kde2 = stats.gaussian_kde(x2, bw_method='silverman')
>>> kde3 = stats.gaussian_kde(x2, bw_method=partial(my_kde_bandwidth, fac=0.2))
>>> kde4 = stats.gaussian_kde(x2, bw_method=partial(my_kde_bandwidth, fac=0.5))
```

```
>>> pdf = stats.norm.pdf
>>> bimodal_pdf = pdf(x_eval, loc=loc1, scale=scale1) * float(size1) / x2.size + \
...               pdf(x_eval, loc=loc2, scale=scale2) * float(size2) / x2.size
```

```
>>> fig = plt.figure(figsize=(8, 6))
>>> ax = fig.add_subplot(111)
```

```
>>> ax.plot(x2, np.zeros(x2.shape), 'b+', ms=12)
>>> ax.plot(x_eval, kde(x_eval), 'k-', label="Scott's Rule")
>>> ax.plot(x_eval, kde2(x_eval), 'b-', label="Silverman's Rule")
>>> ax.plot(x_eval, kde3(x_eval), 'g-', label="Scott * 0.2")
>>> ax.plot(x_eval, kde4(x_eval), 'c-', label="Scott * 0.5")
>>> ax.plot(x_eval, bimodal_pdf, 'r--', label="Actual PDF")
```

```
>>> ax.set_xlim([x_eval.min(), x_eval.max()])
>>> ax.legend(loc=2)
>>> ax.set_xlabel('x')
>>> ax.set_ylabel('Density')
>>> plt.show()
```



As expected, the KDE is not as close to the true PDF as we would like due to the different characteristic size of the two features of the bimodal distribution. By halving the default bandwidth (`Scott * 0.5`) we can do somewhat better, while using a factor 5 smaller bandwidth than the default doesn't smooth enough. What we really need though in this case is a non-uniform (adaptive) bandwidth.

Multivariate estimation

With `gaussian_kde` we can perform multivariate as well as univariate estimation. We demonstrate the bivariate case. First we generate some random data with a model in which the two variates are correlated.

```
>>> def measure(n):
...     """Measurement model, return two coupled measurements."""
...     m1 = np.random.normal(size=n)
...     m2 = np.random.normal(scale=0.5, size=n)
...     return m1+m2, m1-m2
```

```
>>> m1, m2 = measure(2000)
>>> xmin = m1.min()
>>> xmax = m1.max()
>>> ymin = m2.min()
>>> ymax = m2.max()
```

Then we apply the KDE to the data:

```
>>> X, Y = np.mgrid[xmin:xmax:100j, ymin:ymax:100j]
>>> positions = np.vstack([X.ravel(), Y.ravel()])
>>> values = np.vstack([m1, m2])
>>> kernel = stats.gaussian_kde(values)
>>> Z = np.reshape(kernel.evaluate(positions).T, X.shape)
```

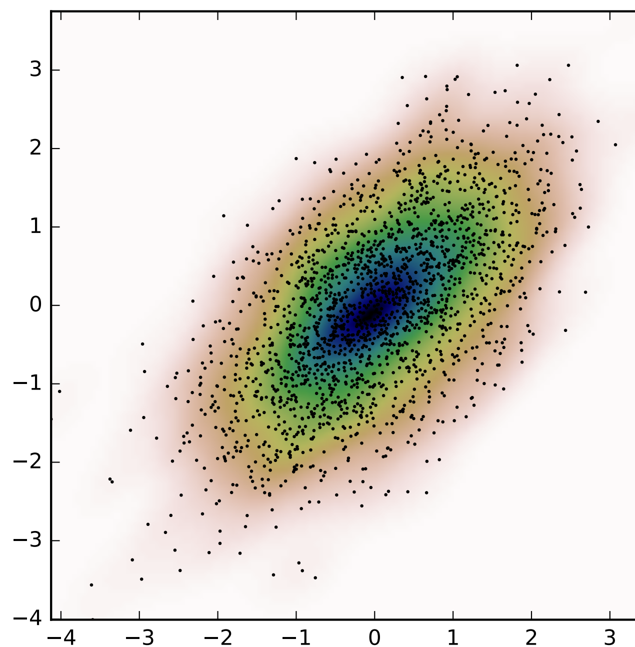
Finally we plot the estimated bivariate distribution as a colormap, and plot the individual data points on top.

```
>>> fig = plt.figure(figsize=(8, 6))
>>> ax = fig.add_subplot(111)
```

```
>>> ax.imshow(np.rot90(Z), cmap=plt.cm.gist_earth_r,
...           extent=[xmin, xmax, ymin, ymax])
>>> ax.plot(m1, m2, 'k.', markersize=2)
```

```
>>> ax.set_xlim([xmin, xmax])
>>> ax.set_ylim([ymin, ymax])
```

```
>>> plt.show()
```



3.1.14 Multidimensional image processing (`scipy.ndimage`)

Introduction

Image processing and analysis are generally seen as operations on two-dimensional arrays of values. There are however a number of fields where images of higher dimensionality must be analyzed. Good examples of these are medical imaging and biological imaging. `numpy` is suited very well for this type of applications due its inherent multidimensional nature. The `scipy.ndimage` packages provides a number of general image processing and analysis functions that are designed to operate with arrays of arbitrary dimensionality. The packages currently includes functions for linear and non-linear filtering, binary morphology, B-spline interpolation, and object measurements.

Properties shared by all functions

All functions share some common properties. Notably, all functions allow the specification of an output array with the *output* argument. With this argument you can specify an array that will be changed in-place with the result with the operation. In this case the result is not returned. Usually, using the *output* argument is more efficient, since an existing array is used to store the result.

The type of arrays returned is dependent on the type of operation, but it is in most cases equal to the type of the input. If, however, the *output* argument is used, the type of the result is equal to the type of the specified output argument. If no output argument is given, it is still possible to specify what the result of the output should be. This is done by simply assigning the desired `numpy` type object to the output argument. For example:

```
>>> from scipy.ndimage import correlate
>>> correlate(np.arange(10), [1, 2.5])
array([ 0,  2,  6,  9, 13, 16, 20, 23, 27, 30])
>>> correlate(np.arange(10), [1, 2.5], output=np.float64)
array([  0. ,   2.5,   6. ,   9.5,  13. ,  16.5,  20. ,  23.5,  27. ,  30.5])
```

Filter functions

The functions described in this section all perform some type of spatial filtering of the input array: the elements in the output are some function of the values in the neighborhood of the corresponding input element. We refer to this neighborhood of elements as the filter kernel, which is often rectangular in shape but may also have an arbitrary footprint. Many of the functions described below allow you to define the footprint of the kernel, by passing a mask through the *footprint* parameter. For example a cross shaped kernel can be defined as follows:

```
>>> footprint = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
>>> footprint
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

Usually the origin of the kernel is at the center calculated by dividing the dimensions of the kernel shape by two. For instance, the origin of a one-dimensional kernel of length three is at the second element. Take for example the correlation of a one-dimensional array with a filter of length 3 consisting of ones:

```
>>> from scipy.ndimage import correlate1d
>>> a = [0, 0, 0, 1, 0, 0, 0]
>>> correlate1d(a, [1, 1, 1])
array([0, 0, 1, 1, 1, 0, 0])
```

Sometimes it is convenient to choose a different origin for the kernel. For this reason most functions support the *origin* parameter which gives the origin of the filter relative to its center. For example:

```
>>> a = [0, 0, 0, 1, 0, 0, 0]
>>> correlate1d(a, [1, 1, 1], origin = -1)
array([0, 1, 1, 1, 0, 0, 0])
```

The effect is a shift of the result towards the left. This feature will not be needed very often, but it may be useful especially for filters that have an even size. A good example is the calculation of backward and forward differences:

```
>>> a = [0, 0, 1, 1, 1, 0, 0]
>>> correlate1d(a, [-1, 1])               # backward difference
array([ 0,  0,  1,  0,  0, -1,  0])
>>> correlate1d(a, [-1, 1], origin = -1)  # forward difference
array([ 0,  1,  0,  0, -1,  0,  0])
```

We could also have calculated the forward difference as follows:

```
>>> correlate1d(a, [0, -1, 1])
array([ 0,  1,  0,  0, -1,  0,  0])
```

However, using the origin parameter instead of a larger kernel is more efficient. For multidimensional kernels *origin* can be a number, in which case the origin is assumed to be equal along all axes, or a sequence giving the origin along each axis.

Since the output elements are a function of elements in the neighborhood of the input elements, the borders of the array need to be dealt with appropriately by providing the values outside the borders. This is done by assuming that the arrays are extended beyond their boundaries according certain boundary conditions. In the functions described below, the boundary conditions can be selected using the *mode* parameter which must be a string with the name of the boundary condition. The following boundary conditions are currently supported:

"nearest"	Use the value at the boundary	[1 2 3]->[1 1 2 3 3]
"wrap"	Periodically replicate the array	[1 2 3]->[3 1 2 3 1]
"reflect"	Reflect the array at the boundary	[1 2 3]->[1 1 2 3 3]
"constant"	Use a constant value, default is 0.0	[1 2 3]->[0 1 2 3 0]

The "constant" mode is special since it needs an additional parameter to specify the constant value that should be used.

Note: The easiest way to implement such boundary conditions would be to copy the data to a larger array and extend the data at the borders according to the boundary conditions. For large arrays and large filter kernels, this would be very memory consuming, and the functions described below therefore use a different approach that does not require allocating large temporary buffers.

Correlation and convolution

- The `correlate1d` function calculates a one-dimensional correlation along the given axis. The lines of the array along the given axis are correlated with the given *weights*. The *weights* parameter must be a one-dimensional sequences of numbers.
- The function `correlate` implements multidimensional correlation of the input array with a given kernel.
- The `convolve1d` function calculates a one-dimensional convolution along the given axis. The lines of the array along the given axis are convoluted with the given *weights*. The *weights* parameter must be a one-dimensional sequences of numbers.

  Note: A convolution is essentially a correlation after mirroring the kernel. As a result, the *origin* parameter behaves differently than in the case of a correlation: the result is shifted in the opposite directions.

- The function `convolve` implements multidimensional convolution of the input array with a given kernel.

Note: A convolution is essentially a correlation after mirroring the kernel. As a result, the *origin* parameter behaves differently than in the case of a correlation: the results is shifted in the opposite direction.

Smoothing filters

- The `gaussian_filter1d` function implements a one-dimensional Gaussian filter. The standard-deviation of the Gaussian filter is passed through the parameter *sigma*. Setting *order* = 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented.
- The `gaussian_filter` function implements a multidimensional Gaussian filter. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions. The order of the filter can be specified separately for each axis. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented. The *order* parameter must be a number, to specify the same order for all axes, or a sequence of numbers to specify a different order for each axis.

Note: The multidimensional filter is implemented as a sequence of one-dimensional Gaussian filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a lower precision, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a more precise output type.

- The `uniform_filter1d` function calculates a one-dimensional uniform filter of the given *size* along the given axis.
- The `uniform_filter` implements a multidimensional uniform filter. The sizes of the uniform filter are given for each axis as a sequence of integers by the *size* parameter. If *size* is not a sequence, but a single number, the sizes along all axis are assumed to be equal.

Note: The multidimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a lower precision, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a more precise output type.

Filters based on order statistics

- The `minimum_filter1d` function calculates a one-dimensional minimum filter of given *size* along the given axis.
- The `maximum_filter1d` function calculates a one-dimensional maximum filter of given *size* along the given axis.
- The `minimum_filter` function calculates a multidimensional minimum filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.
- The `maximum_filter` function calculates a multidimensional maximum filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

- The `rank_filter` function calculates a multidimensional rank filter. The *rank* may be less then zero, i.e., *rank* = -1 indicates the largest element. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.
- The `percentile_filter` function calculates a multidimensional percentile filter. The *percentile* may be less then zero, i.e., *percentile* = -20 equals *percentile* = 80. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.
- The `median_filter` function calculates a multidimensional median filter. Either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint* if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Derivatives

Derivative filters can be constructed in several ways. The function `gaussian_filter1d` described in *Smoothing filters* can be used to calculate derivatives along a given axis using the *order* parameter. Other derivative filters are the Prewitt and Sobel filters:

- The `prewitt` function calculates a derivative along the given axis.
- The `sobel` function calculates a derivative along the given axis.

The Laplace filter is calculated by the sum of the second derivatives along all axes. Thus, different Laplace filters can be constructed using different second derivative functions. Therefore we provide a general function that takes a function argument to calculate the second derivative along a given direction.

- The function `generic_laplace` calculates a laplace filter using the function passed through `derivative2` to calculate second derivatives. The function `derivative2` should have the following signature

```
derivative2(input, axis, output, mode, cval, *extra_arguments, **extra_keywords)
```

  It should calculate the second derivative along the dimension *axis*. If *output* is not `None` it should use that for the output and return None, otherwise it should return the result. *mode*, *cval* have the usual meaning.

  The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and a dictionary of named arguments that are passed to `derivative2` at each call.

  For example

```
>>> def d2(input, axis, output, mode, cval):
...     return correlate1d(input, [1, -2, 1], axis, output, mode, cval, 0)
...
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> from scipy.ndimage import generic_laplace
>>> generic_laplace(a, d2)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1., -4.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

  To demonstrate the use of the *extra_arguments* argument we could do

```
>>> def d2(input, axis, output, mode, cval, weights):
...     return correlate1d(input, weights, axis, output, mode, cval, 0,)
...
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> generic_laplace(a, d2, extra_arguments = ([1, -2, 1],))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1., -4.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

or

```
>>> generic_laplace(a, d2, extra_keywords = {'weights': [1, -2, 1]})
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1., -4.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

The following two functions are implemented using `generic_laplace` by providing appropriate functions for the second derivative function:

- The function `laplace` calculates the Laplace using discrete differentiation for the second derivative (i.e. convolution with `[1, -2, 1]`).
- The function `gaussian_laplace` calculates the Laplace filter using `gaussian_filter` to calculate the second derivatives. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions.

The gradient magnitude is defined as the square root of the sum of the squares of the gradients in all directions. Similar to the generic Laplace function there is a `generic_gradient_magnitude` function that calculats the gradient magnitude of an array.

- The function `generic_gradient_magnitude` calculates a gradient magnitude using the function passed through `derivative` to calculate first derivatives. The function `derivative` should have the following signature

```
derivative(input, axis, output, mode, cval, *extra_arguments, **extra_keywords)
```

It should calculate the derivative along the dimension *axis*. If *output* is not None it should use that for the output and return None, otherwise it should return the result. *mode*, *cval* have the usual meaning.

The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and a dictionary of named arguments that are passed to *derivative* at each call.

For example, the `sobel` function fits the required signature

```
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> from scipy.ndimage import sobel, generic_gradient_magnitude
>>> generic_gradient_magnitude(a, sobel)
array([[ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  1.41421356,  2.        ,  1.41421356,  0.        ],
       [ 0.        ,  2.        ,  0.        ,  2.        ,  0.        ],
       [ 0.        ,  1.41421356,  2.        ,  1.41421356,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ]])
```

See the documentation of *generic_laplace* for examples of using the *extra_arguments* and *extra_keywords* arguments.

The *sobel* and *prewitt* functions fit the required signature and can therefore directly be used with *generic_gradient_magnitude*.

- The function *gaussian_gradient_magnitude* calculates the gradient magnitude using *gaussian_filter* to calculate the first derivatives. The standard-deviations of the Gaussian filter along each axis are passed through the parameter *sigma* as a sequence or numbers. If *sigma* is not a sequence but a single number, the standard deviation of the filter is equal along all directions.

Generic filter functions

To implement filter functions, generic functions can be used that accept a callable object that implements the filtering operation. The iteration over the input and output arrays is handled by these generic functions, along with such details as the implementation of the boundary conditions. Only a callable object implementing a callback function that does the actual filtering work must be provided. The callback function can also be written in C and passed using a PyCapsule (see *Extending scipy.ndimage in C* for more information).

- The *generic_filter1d* function implements a generic one-dimensional filter function, where the actual filtering operation must be supplied as a python function (or other callable object). The *generic_filter1d* function iterates over the lines of an array and calls `function` at each line. The arguments that are passed to `function` are one-dimensional arrays of the `tFloat64` type. The first contains the values of the current line. It is extended at the beginning end the end, according to the *filter_size* and *origin* arguments. The second array should be modified in-place to provide the output values of the line. For example consider a correlation along one dimension:

```
>>> a = np.arange(12).reshape(3,4)
>>> correlate1d(a, [1, 2, 3])
array([[ 3,  8, 14, 17],
       [27, 32, 38, 41],
       [51, 56, 62, 65]])
```

The same operation can be implemented using *generic_filter1d* as follows:

```
>>> def fnc(iline, oline):
...     oline[...] = iline[:-2] + 2 * iline[1:-1] + 3 * iline[2:]
...
>>> from scipy.ndimage import generic_filter1d
>>> generic_filter1d(a, fnc, 3)
array([[ 3,  8, 14, 17],
       [27, 32, 38, 41],
       [51, 56, 62, 65]])
```

Here the origin of the kernel was (by default) assumed to be in the middle of the filter of length 3. Therefore, each input line was extended by one value at the beginning and at the end, before the function was called.

Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the parameters of our filter as an argument

```
>>> def fnc(iline, oline, a, b):
...     oline[...] = iline[:-2] + a * iline[1:-1] + b * iline[2:]
...
>>> generic_filter1d(a, fnc, 3, extra_arguments = (2, 3))
array([[ 3,  8, 14, 17],
       [27, 32, 38, 41],
       [51, 56, 62, 65]])
```

or

```
>>> generic_filter1d(a, fnc, 3, extra_keywords = {'a':2, 'b':3})
array([[ 3,  8, 14, 17],
       [27, 32, 38, 41],
       [51, 56, 62, 65]])
```

- The `generic_filter` function implements a generic filter function, where the actual filtering operation must be supplied as a python function (or other callable object). The `generic_filter` function iterates over the array and calls `function` at each element. The argument of `function` is a one-dimensional array of the `tFloat64` type, that contains the values around the current element that are within the footprint of the filter. The function should return a single value that can be converted to a double precision number. For example consider a correlation:

```
>>> a = np.arange(12).reshape(3,4)
>>> correlate(a, [[1, 0], [0, 3]])
array([[ 0,  3,  7, 11],
       [12, 15, 19, 23],
       [28, 31, 35, 39]])
```

The same operation can be implemented using *generic_filter* as follows:

```
>>> def fnc(buffer):
...     return (buffer * np.array([1, 3])).sum()
...
>>> from scipy.ndimage import generic_filter
>>> generic_filter(a, fnc, footprint = [[1, 0], [0, 1]])
array([[ 0,  3,  7, 11],
       [12, 15, 19, 23],
       [28, 31, 35, 39]])
```

Here a kernel footprint was specified that contains only two elements. Therefore the filter function receives a buffer of length equal to two, which was multiplied with the proper weights and the result summed.

When calling `generic_filter`, either the sizes of a rectangular kernel or the footprint of the kernel must be provided. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint*, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the parameters of our filter as an argument

```
>>> def fnc(buffer, weights):
...     weights = np.asarray(weights)
...     return (buffer * weights).sum()
...
>>> generic_filter(a, fnc, footprint = [[1, 0], [0, 1]], extra_arguments = ([1,␣
↪3],))
array([[ 0,  3,  7, 11],
       [12, 15, 19, 23],
       [28, 31, 35, 39]])
```

or

```
>>> generic_filter(a, fnc, footprint = [[1, 0], [0, 1]], extra_keywords= {'weights
↪': [1, 3]})
array([[ 0,  3,  7, 11],
```

```
       [12, 15, 19, 23],
       [28, 31, 35, 39]])
```

These functions iterate over the lines or elements starting at the last axis, i.e. the last index changes the fastest. This order of iteration is guaranteed for the case that it is important to adapt the filter depending on spatial location. Here is an example of using a class that implements the filter and keeps track of the current coordinates while iterating. It performs the same filter operation as described above for `generic_filter`, but additionally prints the current coordinates:

```
>>> a = np.arange(12).reshape(3,4)
>>>
>>> class fnc_class:
...     def __init__(self, shape):
...         # store the shape:
...         self.shape = shape
...         # initialize the coordinates:
...         self.coordinates = [0] * len(shape)
...
...     def filter(self, buffer):
...         result = (buffer * np.array([1, 3])).sum()
...         print self.coordinates
...         # calculate the next coordinates:
...         axes = range(len(self.shape))
...         axes.reverse()
...         for jj in axes:
...             if self.coordinates[jj] < self.shape[jj] - 1:
...                 self.coordinates[jj] += 1
...                 break
...             else:
...                 self.coordinates[jj] = 0
...         return result
...
>>> fnc = fnc_class(shape = (3,4))
>>> generic_filter(a, fnc.filter, footprint = [[1, 0], [0, 1]])
[0, 0]
[0, 1]
[0, 2]
[0, 3]
[1, 0]
[1, 1]
[1, 2]
[1, 3]
[2, 0]
[2, 1]
[2, 2]
[2, 3]
array([[ 0,  3,  7, 11],
       [12, 15, 19, 23],
       [28, 31, 35, 39]])
```

For the `generic_filter1d` function the same approach works, except that this function does not iterate over the axis that is being filtered. The example for `generic_filter1d` then becomes this:

```
>>> a = np.arange(12).reshape(3,4)
>>>
>>> class fnc1d_class:
...     def __init__(self, shape, axis = -1):
```

```
...         # store the filter axis:
...         self.axis = axis
...         # store the shape:
...         self.shape = shape
...         # initialize the coordinates:
...         self.coordinates = [0] * len(shape)
...
...     def filter(self, iline, oline):
...         oline[...] = iline[:-2] + 2 * iline[1:-1] + 3 * iline[2:]
...         print self.coordinates
...         # calculate the next coordinates:
...         axes = range(len(self.shape))
...         # skip the filter axis:
...         del axes[self.axis]
...         axes.reverse()
...         for jj in axes:
...             if self.coordinates[jj] < self.shape[jj] - 1:
...                 self.coordinates[jj] += 1
...                 break
...             else:
...                 self.coordinates[jj] = 0
...
>>> fnc = fnc1d_class(shape = (3,4))
>>> generic_filter1d(a, fnc.filter, 3)
[0, 0]
[1, 0]
[2, 0]
array([[ 3,  8, 14, 17],
       [27, 32, 38, 41],
       [51, 56, 62, 65]])
```

Fourier domain filters

The functions described in this section perform filtering operations in the Fourier domain. Thus, the input array of such a function should be compatible with an inverse Fourier transform function, such as the functions from the `numpy.fft` module. We therefore have to deal with arrays that may be the result of a real or a complex Fourier transform. In the case of a real Fourier transform only half of the of the symmetric complex transform is stored. Additionally, it needs to be known what the length of the axis was that was transformed by the real fft. The functions described here provide a parameter *n* that in the case of a real transform must be equal to the length of the real transform axis before transformation. If this parameter is less than zero, it is assumed that the input array was the result of a complex Fourier transform. The parameter *axis* can be used to indicate along which axis the real transform was executed.

- The *fourier_shift* function multiplies the input array with the multidimensional Fourier transform of a shift operation for the given shift. The *shift* parameter is a sequences of shifts for each dimension, or a single value for all dimensions.
- The *fourier_gaussian* function multiplies the input array with the multidimensional Fourier transform of a Gaussian filter with given standard-deviations *sigma*. The *sigma* parameter is a sequences of values for each dimension, or a single value for all dimensions.
- The *fourier_uniform* function multiplies the input array with the multidimensional Fourier transform of a uniform filter with given sizes *size*. The *size* parameter is a sequences of values for each dimension, or a single value for all dimensions.
- The *fourier_ellipsoid* function multiplies the input array with the multidimensional Fourier transform of a elliptically shaped filter with given sizes *size*. The *size* parameter is a sequences of values for each dimension, or a single value for all dimensions. This function is only implemented for dimensions 1, 2, and 3.

Interpolation functions

This section describes various interpolation functions that are based on B-spline theory. A good introduction to B-splines can be found in[1].

Spline pre-filters

Interpolation using splines of an order larger than 1 requires a pre-filtering step. The interpolation functions described in section *Interpolation functions* apply pre-filtering by calling `spline_filter`, but they can be instructed not to do this by setting the *prefilter* keyword equal to False. This is useful if more than one interpolation operation is done on the same array. In this case it is more efficient to do the pre-filtering only once and use a prefiltered array as the input of the interpolation functions. The following two functions implement the pre-filtering:

- The `spline_filter1d` function calculates a one-dimensional spline filter along the given axis. An output array can optionally be provided. The order of the spline must be larger then 1 and less than 6.
- The `spline_filter` function calculates a multidimensional spline filter.

Note: The multidimensional filter is implemented as a sequence of one-dimensional spline filters. The intermediate arrays are stored in the same data type as the output. Therefore, if an output with a limited precision is requested, the results may be imprecise because intermediate results may be stored with insufficient precision. This can be prevented by specifying a output type of high precision.

Interpolation functions

Following functions all employ spline interpolation to effect some type of geometric transformation of the input array. This requires a mapping of the output coordinates to the input coordinates, and therefore the possibility arises that input values outside the boundaries are needed. This problem is solved in the same way as described in *Filter functions* for the multidimensional filter functions. Therefore these functions all support a *mode* parameter that determines how the boundaries are handled, and a *cval* parameter that gives a constant value in case that the 'constant' mode is used.

- The `geometric_transform` function applies an arbitrary geometric transform to the input. The given *mapping* function is called at each point in the output to find the corresponding coordinates in the input. *mapping* must be a callable object that accepts a tuple of length equal to the output array rank and returns the corresponding input coordinates as a tuple of length equal to the input array rank. The output shape and output type can optionally be provided. If not given they are equal to the input shape and type.

  For example:

```
>>> a = np.arange(12).reshape(4,3).astype(np.float64)
>>> def shift_func(output_coordinates):
...     return (output_coordinates[0] - 0.5, output_coordinates[1] - 0.5)
...
>>> from scipy.ndimage import geometric_transform
>>> geometric_transform(a, shift_func)
array([[ 0.    ,  0.    ,  0.    ],
       [ 0.    ,  1.3625,  2.7375],
       [ 0.    ,  4.8125,  6.1875],
       [ 0.    ,  8.2625,  9.6375]])
```

  Optionally extra arguments can be defined and passed to the filter function. The *extra_arguments* and *extra_keywords* arguments can be used to pass a tuple of extra arguments and/or a dictionary of named arguments that are passed to derivative at each call. For example, we can pass the shifts in our example as arguments

[1] M. Unser, "Splines: A Perfect Fit for Signal and Image Processing," IEEE Signal Processing Magazine, vol. 16, no. 6, pp. 22-38, November 1999.

```
>>> def shift_func(output_coordinates, s0, s1):
...     return (output_coordinates[0] - s0, output_coordinates[1] - s1)
...
>>> geometric_transform(a, shift_func, extra_arguments = (0.5, 0.5))
array([[ 0.    ,  0.    ,  0.    ],
       [ 0.    ,  1.3625,  2.7375],
       [ 0.    ,  4.8125,  6.1875],
       [ 0.    ,  8.2625,  9.6375]])
```

or

```
>>> geometric_transform(a, shift_func, extra_keywords = {'s0': 0.5, 's1': 0.5})
array([[ 0.    ,  0.    ,  0.    ],
       [ 0.    ,  1.3625,  2.7375],
       [ 0.    ,  4.8125,  6.1875],
       [ 0.    ,  8.2625,  9.6375]])
```

Note: The mapping function can also be written in C and passed using a `scipy.LowLevelCallable`. See *Extending scipy.ndimage in C* for more information.

- The function `map_coordinates` applies an arbitrary coordinate transformation using the given array of coordinates. The shape of the output is derived from that of the coordinate array by dropping the first axis. The parameter *coordinates* is used to find for each point in the output the corresponding coordinates in the input. The values of *coordinates* along the first axis are the coordinates in the input array at which the output value is found. (See also the numarray *coordinates* function.) Since the coordinates may be non- integer coordinates, the value of the input at these coordinates is determined by spline interpolation of the requested order.

  Here is an example that interpolates a 2D array at `(0.5, 0.5)` and `(1, 2)`:

```
>>> a = np.arange(12).reshape(4,3).astype(np.float64)
>>> a
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
>>> from scipy.ndimage import map_coordinates
>>> map_coordinates(a, [[0.5, 2], [0.5, 1]])
array([ 1.3625,  7.])
```

- The `affine_transform` function applies an affine transformation to the input array. The given transformation *matrix* and *offset* are used to find for each point in the output the corresponding coordinates in the input. The value of the input at the calculated coordinates is determined by spline interpolation of the requested order. The transformation *matrix* must be two-dimensional or can also be given as a one-dimensional sequence or array. In the latter case, it is assumed that the matrix is diagonal. A more efficient interpolation algorithm is then applied that exploits the separability of the problem. The output shape and output type can optionally be provided. If not given they are equal to the input shape and type.
- The `shift` function returns a shifted version of the input, using spline interpolation of the requested *order*.
- The `zoom` function returns a rescaled version of the input, using spline interpolation of the requested *order*.
- The `rotate` function returns the input array rotated in the plane defined by the two axes given by the parameter *axes*, using spline interpolation of the requested *order*. The angle must be given in degrees. If *reshape* is true, then the size of the output array is adapted to contain the rotated input.

Morphology

Binary morphology

- The `generate_binary_structure` functions generates a binary structuring element for use in binary morphology operations. The *rank* of the structure must be provided. The size of the structure that is returned is equal to three in each direction. The value of each element is equal to one if the square of the Euclidean distance from the element to the center is less or equal to *connectivity*. For instance, two dimensional 4-connected and 8-connected structures are generated as follows:

```
>>> from scipy.ndimage import generate_binary_structure
>>> generate_binary_structure(2, 1)
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> generate_binary_structure(2, 2)
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

Most binary morphology functions can be expressed in terms of the basic operations erosion and dilation.

- The `binary_erosion` function implements binary erosion of arrays of arbitrary rank with the given structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *border_value* parameter gives the value of the array outside boundaries. The erosion is repeated *iterations* times. If *iterations* is less than one, the erosion is repeated until the result does not change anymore. If a *mask* array is given, only those elements with a true value at the corresponding mask element are modified at each iteration.
- The `binary_dilation` function implements binary dilation of arrays of arbitrary rank with the given structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *border_value* parameter gives the value of the array outside boundaries. The dilation is repeated *iterations* times. If *iterations* is less than one, the dilation is repeated until the result does not change anymore. If a *mask* array is given, only those elements with a true value at the corresponding mask element are modified at each iteration.

Here is an example of using `binary_dilation` to find all elements that touch the border, by repeatedly dilating an empty array from the border using the data array as the mask:

```
>>> struct = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
>>> a = np.array([[1,0,0,0,0], [1,1,0,1,0], [0,0,1,1,0], [0,0,0,0,0]])
>>> a
array([[1, 0, 0, 0, 0],
       [1, 1, 0, 1, 0],
       [0, 0, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> from scipy.ndimage import binary_dilation
>>> binary_dilation(np.zeros(a.shape), struct, -1, a, border_value=1)
array([[ True, False, False, False, False],
       [ True,  True, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False]], dtype=bool)
```

The `binary_erosion` and `binary_dilation` functions both have an *iterations* parameter which allows the erosion or dilation to be repeated a number of times. Repeating an erosion or a dilation with a given structure *n* times

is equivalent to an erosion or a dilation with a structure that is *n-1* times dilated with itself. A function is provided that allows the calculation of a structure that is dilated a number of times with itself:

- The `iterate_structure` function returns a structure by dilation of the input structure *iteration* - 1 times with itself.

  For instance:

```
>>> struct = generate_binary_structure(2, 1)
>>> struct
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> from scipy.ndimage import iterate_structure
>>> iterate_structure(struct, 2)
array([[False, False,  True, False, False],
       [False,  True,  True,  True, False],
       [ True,  True,  True,  True,  True],
       [False,  True,  True,  True, False],
       [False, False,  True, False, False]], dtype=bool)

If the origin of the original structure is equal to 0, then it is
also equal to 0 for the iterated structure. If not, the origin
must also be adapted if the equivalent of the *iterations*
erosions or dilations must be achieved with the iterated
structure. The adapted origin is simply obtained by multiplying
with the number of iterations. For convenience the
:func:`iterate_structure` also returns the adapted origin if the
*origin* parameter is not ``None``:

.. code:: python

   >>> iterate_structure(struct, 2, -1)
   (array([[False, False,  True, False, False],
           [False,  True,  True,  True, False],
           [ True,  True,  True,  True,  True],
           [False,  True,  True,  True, False],
           [False, False,  True, False, False]], dtype=bool), [-2, -2])
```

Other morphology operations can be defined in terms of erosion and d dilation. The following functions provide a few of these operations for convenience:

- The `binary_opening` function implements binary opening of arrays of arbitrary rank with the given structuring element. Binary opening is equivalent to a binary erosion followed by a binary dilation with the same structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *iterations* parameter gives the number of erosions that is performed followed by the same number of dilations.
- The `binary_closing` function implements binary closing of arrays of arbitrary rank with the given structuring element. Binary closing is equivalent to a binary dilation followed by a binary erosion with the same structuring element. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is generated using `generate_binary_structure`. The *iterations* parameter gives the number of dilations that is performed followed by the same number of erosions.
- The `binary_fill_holes` function is used to close holes in objects in a binary image, where the structure defines the connectivity of the holes. The origin parameter controls the placement of the structuring element as described in *Filter functions*. If no structuring element is provided, an element with connectivity equal to one is

generated using `generate_binary_structure`.

- The `binary_hit_or_miss` function implements a binary hit-or-miss transform of arrays of arbitrary rank with the given structuring elements. The hit-or-miss transform is calculated by erosion of the input with the first structure, erosion of the logical *not* of the input with the second structure, followed by the logical *and* of these two erosions. The origin parameters control the placement of the structuring elements as described in *Filter functions*. If *origin2* equals None it is set equal to the *origin1* parameter. If the first structuring element is not provided, a structuring element with connectivity equal to one is generated using `generate_binary_structure`, if *structure2* is not provided, it is set equal to the logical *not* of *structure1*.

Grey-scale morphology

Grey-scale morphology operations are the equivalents of binary morphology operations that operate on arrays with arbitrary values. Below we describe the grey-scale equivalents of erosion, dilation, opening and closing. These operations are implemented in a similar fashion as the filters described in *Filter functions*, and we refer to this section for the description of filter kernels and footprints, and the handling of array borders. The grey-scale morphology operations optionally take a *structure* parameter that gives the values of the structuring element. If this parameter is not given the structuring element is assumed to be flat with a value equal to zero. The shape of the structure can optionally be defined by the *footprint* parameter. If this parameter is not given, the structure is assumed to be rectangular, with sizes equal to the dimensions of the *structure* array, or by the *size* parameter if *structure* is not given. The *size* parameter is only used if both *structure* and *footprint* are not given, in which case the structuring element is assumed to be rectangular and flat with the dimensions given by *size*. The *size* parameter, if provided, must be a sequence of sizes or a single number in which case the size of the filter is assumed to be equal along each axis. The *footprint* parameter, if provided, must be an array that defines the shape of the kernel by its non-zero elements.

Similar to binary erosion and dilation there are operations for grey-scale erosion and dilation:

- The `grey_erosion` function calculates a multidimensional grey- scale erosion.
- The `grey_dilation` function calculates a multidimensional grey-scale dilation.

Grey-scale opening and closing operations can be defined similar to their binary counterparts:

- The `grey_opening` function implements grey-scale opening of arrays of arbitrary rank. Grey-scale opening is equivalent to a grey-scale erosion followed by a grey-scale dilation.
- The `grey_closing` function implements grey-scale closing of arrays of arbitrary rank. Grey-scale opening is equivalent to a grey-scale dilation followed by a grey-scale erosion.
- The `morphological_gradient` function implements a grey-scale morphological gradient of arrays of arbitrary rank. The grey-scale morphological gradient is equal to the difference of a grey-scale dilation and a grey-scale erosion.
- The `morphological_laplace` function implements a grey-scale morphological laplace of arrays of arbitrary rank. The grey-scale morphological laplace is equal to the sum of a grey-scale dilation and a grey-scale erosion minus twice the input.
- The `white_tophat` function implements a white top-hat filter of arrays of arbitrary rank. The white top-hat is equal to the difference of the input and a grey-scale opening.
- The `black_tophat` function implements a black top-hat filter of arrays of arbitrary rank. The black top-hat is equal to the difference of a grey-scale closing and the input.

Distance transforms

Distance transforms are used to calculate the minimum distance from each element of an object to the background. The following functions implement distance transforms for three different distance metrics: Euclidean, City Block, and Chessboard distances.

- The function *distance_transform_cdt* uses a chamfer type algorithm to calculate the distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest distance to the background (all non-object elements). The structure determines the type of chamfering that is done. If the structure is equal to 'cityblock' a structure is generated using *generate_binary_structure* with a squared distance equal to 1. If the structure is equal to 'chessboard', a structure is generated using *generate_binary_structure* with a squared distance equal to the rank of the array. These choices correspond to the common interpretations of the cityblock and the chessboard distance metrics in two dimensions.

  In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

  The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (both `Int32`). The basics of the algorithm used to implement this function is described in[2].

- The function *distance_transform_edt* calculates the exact euclidean distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest euclidean distance to the background (all non-object elements).

  In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

  Optionally the sampling along each axis can be given by the *sampling* parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes.

  The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (`Float64` and `Int32`).The algorithm used to implement this function is described in[3].

- The function *distance_transform_bf* uses a brute-force algorithm to calculate the distance transform of the input, by replacing each object element (defined by values larger than zero) with the shortest distance to the background (all non-object elements). The metric must be one of "euclidean", "cityblock", or "chessboard".

  In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result. The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.

  Optionally the sampling along each axis can be given by the *sampling* parameter which should be a sequence of length equal to the input rank, or a single number in which the sampling is assumed to be equal along all axes. This parameter is only used in the case of the euclidean distance transform.

  The *distances* and *indices* arguments can be used to give optional output arrays that must be of the correct size and type (`Float64` and `Int32`).

  Note: This function uses a slow brute-force algorithm, the function *distance_transform_cdt* can be used to more efficiently calculate cityblock and chessboard distance transforms. The function *distance_transform_edt* can be used to more efficiently calculate the exact euclidean distance transform.

Segmentation and labeling

Segmentation is the process of separating objects of interest from the background. The most simple approach is probably intensity thresholding, which is easily done with `numpy` functions:

[2] G. Borgefors, "Distance transformations in arbitrary dimensions.", Computer Vision, Graphics, and Image Processing, 27:321-345, 1984.

[3] C. R. Maurer, Jr., R. Qi, and V. Raghavan, "A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. IEEE Trans. PAMI 25, 265-270, 2003.

```
>>> a = np.array([[1,2,2,1,1,0],
...               [0,2,3,1,2,0],
...               [1,1,1,3,3,2],
...               [1,1,1,1,2,1]])
>>> np.where(a > 1, 1, 0)
array([[0, 1, 1, 0, 0, 0],
       [0, 1, 1, 0, 1, 0],
       [0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 1, 0]])
```

The result is a binary image, in which the individual objects still need to be identified and labeled. The function *label* generates an array where each object is assigned a unique number:

- The *label* function generates an array where the objects in the input are labeled with an integer index. It returns a tuple consisting of the array of object labels and the number of objects found, unless the *output* parameter is given, in which case only the number of objects is returned. The connectivity of the objects is defined by a structuring element. For instance, in two dimensions using a four-connected structuring element gives:

```
>>> a = np.array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> s = [[0, 1, 0], [1,1,1], [0,1,0]]
>>> from scipy.ndimage import label
>>> label(a, s)
(array([[0, 1, 1, 0, 0, 0],
        [0, 1, 1, 0, 2, 0],
        [0, 0, 0, 2, 2, 2],
        [0, 0, 0, 0, 2, 0]]), 2)
```

These two objects are not connected because there is no way in which we can place the structuring element such that it overlaps with both objects. However, an 8-connected structuring element results in only a single object:

```
>>> a = np.array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> s = [[1,1,1], [1,1,1], [1,1,1]]
>>> label(a, s)[0]
array([[0, 1, 1, 0, 0, 0],
       [0, 1, 1, 0, 1, 0],
       [0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 1, 0]])
```

If no structuring element is provided, one is generated by calling *generate_binary_structure* (see *Binary morphology*) using a connectivity of one (which in 2D is the 4-connected structure of the first example). The input can be of any type, any value not equal to zero is taken to be part of an object. This is useful if you need to 're-label' an array of object indices, for instance after removing unwanted objects. Just apply the label function again to the index array. For instance:

```
>>> l, n = label([1, 0, 1, 0, 1])
>>> l
array([1, 0, 2, 0, 3])
>>> l = np.where(l != 2, l, 0)
>>> l
array([1, 0, 0, 0, 3])
>>> label(l)[0]
array([1, 0, 0, 0, 2])
```

Note: The structuring element used by *label* is assumed to be symmetric.

There is a large number of other approaches for segmentation, for instance from an estimation of the borders of the objects that can be obtained for instance by derivative filters. One such an approach is watershed segmentation. The function `watershed_ift` generates an array where each object is assigned a unique label, from an array that localizes the object borders, generated for instance by a gradient magnitude filter. It uses an array containing initial markers for the objects:

- The `watershed_ift` function applies a watershed from markers algorithm, using an Iterative Forest Transform, as described in[4].
- The inputs of this function are the array to which the transform is applied, and an array of markers that designate the objects by a unique label, where any non-zero value is a marker. For instance:

```
>>> input = np.array([[0, 0, 0, 0, 0, 0, 0],
...                   [0, 1, 1, 1, 1, 1, 0],
...                   [0, 1, 0, 0, 0, 1, 0],
...                   [0, 1, 0, 0, 0, 1, 0],
...                   [0, 1, 0, 0, 0, 1, 0],
...                   [0, 1, 1, 1, 1, 1, 0],
...                   [0, 0, 0, 0, 0, 0, 0]], np.uint8)
>>> markers = np.array([[1, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 2, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0]], np.int8)
>>> from scipy.ndimage import watershed_ift
>>> watershed_ift(input, markers)
array([[1, 1, 1, 1, 1, 1, 1],
       [1, 1, 2, 2, 2, 1, 1],
       [1, 2, 2, 2, 2, 2, 1],
       [1, 2, 2, 2, 2, 2, 1],
       [1, 2, 2, 2, 2, 2, 1],
       [1, 1, 2, 2, 2, 1, 1],
       [1, 1, 1, 1, 1, 1, 1]], dtype=int8)
```

Here two markers were used to designate an object (*marker* = 2) and the background (*marker* = 1). The order in which these are processed is arbitrary: moving the marker for the background to the lower right corner of the array yields a different result:

```
>>> markers = np.array([[0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 2, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 1]], np.int8)
>>> watershed_ift(input, markers)
array([[1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1],
       [1, 1, 2, 2, 2, 1, 1],
       [1, 1, 2, 2, 2, 1, 1],
       [1, 1, 2, 2, 2, 1, 1],
       [1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1]], dtype=int8)
```

[4] P. Felkel, R. Wegenkittl, and M. Bruckschwaiger, "Implementation and Complexity of the Watershed-from-Markers Algorithm Computed as a Minimal Cost Forest.", Eurographics 2001, pp. C:26-35.

The result is that the object (*marker* = 2) is smaller because the second marker was processed earlier. This may not be the desired effect if the first marker was supposed to designate a background object. Therefore `watershed_ift` treats markers with a negative value explicitly as background markers and processes them after the normal markers. For instance, replacing the first marker by a negative marker gives a result similar to the first example:

```
>>> markers = np.array([[0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 2, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, 0],
...                     [0, 0, 0, 0, 0, 0, -1]], np.int8)
>>> watershed_ift(input, markers)
array([[-1, -1, -1, -1, -1, -1, -1],
       [-1, -1,  2,  2,  2, -1, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1, -1,  2,  2,  2, -1, -1],
       [-1, -1, -1, -1, -1, -1, -1]], dtype=int8)
```

The connectivity of the objects is defined by a structuring element. If no structuring element is provided, one is generated by calling `generate_binary_structure` (see *Binary morphology*) using a connectivity of one (which in 2D is a 4-connected structure.) For example, using an 8-connected structure with the last example yields a different object:

```
>>> watershed_ift(input, markers,
...               structure = [[1,1,1], [1,1,1], [1,1,1]])
array([[-1, -1, -1, -1, -1, -1, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1,  2,  2,  2,  2,  2, -1],
       [-1, -1, -1, -1, -1, -1, -1]], dtype=int8)
```

Note: The implementation of `watershed_ift` limits the data types of the input to `UInt8` and `UInt16`.

Object measurements

Given an array of labeled objects, the properties of the individual objects can be measured. The `find_objects` function can be used to generate a list of slices that for each object, give the smallest sub-array that fully contains the object:

- The `find_objects` function finds all objects in a labeled array and returns a list of slices that correspond to the smallest regions in the array that contains the object.

  For instance:

```
>>> a = np.array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> l, n = label(a)
>>> from scipy.ndimage import find_objects
>>> f = find_objects(l)
>>> a[f[0]]
```

```
array([[1, 1],
       [1, 1]])
>>> a[f[1]]
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
```

The function *find_objects* returns slices for all objects, unless the *max_label* parameter is larger then zero, in which case only the first *max_label* objects are returned. If an index is missing in the *label* array, `None` is return instead of a slice. For example:

```
>>> from scipy.ndimage import find_objects
>>> find_objects([1, 0, 3, 4], max_label = 3)
[(slice(0, 1, None),), None, (slice(2, 3, None),)]
```

The list of slices generated by *find_objects* is useful to find the position and dimensions of the objects in the array, but can also be used to perform measurements on the individual objects. Say we want to find the sum of the intensities of an object in image:

```
>>> image = np.arange(4 * 6).reshape(4, 6)
>>> mask = np.array([[0,1,1,0,0,0],[0,1,1,0,1,0],[0,0,0,1,1,1],[0,0,0,0,1,0]])
>>> labels = label(mask)[0]
>>> slices = find_objects(labels)
```

Then we can calculate the sum of the elements in the second object:

```
>>> np.where(labels[slices[1]] == 2, image[slices[1]], 0).sum()
80
```

That is however not particularly efficient, and may also be more complicated for other types of measurements. Therefore a few measurements functions are defined that accept the array of object labels and the index of the object to be measured. For instance calculating the sum of the intensities can be done by:

```
>>> from scipy.ndimage import sum as ndi_sum
>>> ndi_sum(image, labels, 2)
80
```

For large arrays and small objects it is more efficient to call the measurement functions after slicing the array:

```
>>> ndi_sum(image[slices[1]], labels[slices[1]], 2)
80
```

Alternatively, we can do the measurements for a number of labels with a single function call, returning a list of results. For instance, to measure the sum of the values of the background and the second object in our example we give a list of labels:

```
>>> ndi_sum(image, labels, [0, 2])
array([178.0, 80.0])
```

The measurement functions described below all support the *index* parameter to indicate which object(s) should be measured. The default value of *index* is None. This indicates that all elements where the label is larger than zero should be treated as a single object and measured. Thus, in this case the *labels* array is treated as a mask defined by the elements that are larger than zero. If *index* is a number or a sequence of numbers it gives the labels of the objects that are measured. If *index* is a sequence, a list of the results is returned. Functions that return more than one result, return their result as a tuple if *index* is a single number, or as a tuple of lists, if *index* is a sequence.

- The `sum` function calculates the sum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `mean` function calculates the mean of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `variance` function calculates the variance of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `standard_deviation` function calculates the standard deviation of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `minimum` function calculates the minimum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `maximum` function calculates the maximum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `minimum_position` function calculates the position of the minimum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `maximum_position` function calculates the position of the maximum of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `extrema` function calculates the minimum, the maximum, and their positions, of the elements of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation. The result is a tuple giving the minimum, the maximum, the position of the minimum and the position of the maximum. The result is the same as a tuple formed by the results of the functions *minimum*, *maximum*, *minimum_position*, and *maximum_position* that are described above.
- The `center_of_mass` function calculates the center of mass of the of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation.
- The `histogram` function calculates a histogram of the of the object with label(s) given by *index*, using the *labels* array for the object labels. If *index* is `None`, all elements with a non-zero label value are treated as a single object. If *label* is `None`, all elements of *input* are used in the calculation. Histograms are defined by their minimum (*min*), maximum (*max*) and the number of bins (*bins*). They are returned as one-dimensional arrays of type `Int32`.

Extending `scipy.ndimage` in C

A few functions in `scipy.ndimage` take a callback argument. This can be either a python function or a `scipy.LowLevelCallable` containing a pointer to a C function. Using a C function will generally be more efficient since it avoids the overhead of calling a python function on many elements of an array. To use a C function you must write a C extension that contains the callback function and a Python function that returns a `scipy.LowLevelCallable` containing a pointer to the callback.

An example of a function that supports callbacks is *geometric_transform*, which accepts a callback function that defines a mapping from all output coordinates to corresponding coordinates in the input array. Consider the following python example which uses *geometric_transform* to implement a shift function.

```
from scipy import ndimage

def transform(output_coordinates, shift):
    input_coordinates = output_coordinates[0] - shift, output_coordinates[1] - shift
    return input_coordinates

im = np.arange(12).reshape(4, 3).astype(np.float64)
shift = 0.5
print(ndimage.geometric_transform(im, transform, extra_arguments=(shift,)))
```

We can also implement the callback function with the following C code.

```
/* example.c */

#include <Python.h>
#include <numpy/npy_common.h>

static int
_transform(npy_intp *output_coordinates, double *input_coordinates,
           int output_rank, int input_rank, void *user_data)
{
    npy_intp i;
    double shift = *(double *)user_data;

    for (i = 0; i < input_rank; i++) {
        input_coordinates[i] = output_coordinates[i] - shift;
    }
    return 1;
}

static char *transform_signature = "int (npy_intp *, double *, int, int, void *)";

static PyObject *
py_get_transform(PyObject *obj, PyObject *args)
{
    if (!PyArg_ParseTuple(args, "")) return NULL;
    return PyCapsule_New(_transform, transform_signature, NULL);
}

static PyMethodDef ExampleMethods[] = {
    {"get_transform", (PyCFunction)py_get_transform, METH_VARARGS, ""},
    {NULL, NULL, 0, NULL}
};

/* Initialize the module */
#if PY_VERSION_HEX >= 0x03000000
static struct PyModuleDef example = {
    PyModuleDef_HEAD_INIT,
    "example",
    NULL,
    -1,
    ExampleMethods,
    NULL,
    NULL,
```

```
    NULL,
    NULL
};

PyMODINIT_FUNC
PyInit_example(void)
{
    return PyModule_Create(&example);
}
#else
PyMODINIT_FUNC
initexample(void)
{
    Py_InitModule("example", ExampleMethods);
}
#endif
```

More information on writing Python extension modules can be found here. If the C code is in the file `example.c`, then it can be compiled with the following `setup.py`,

```
from distutils.core import setup, Extension
import numpy

shift = Extension('example',
                  ['example.c'],
                  include_dirs=[numpy.get_include()]
)

setup(name='example',
      ext_modules=[shift]
)
```

and now running the script

```
import ctypes
import numpy as np
from scipy import ndimage, LowLevelCallable

from example import get_transform

shift = 0.5

user_data = ctypes.c_double(shift)
ptr = ctypes.cast(ctypes.pointer(user_data), ctypes.c_void_p)
callback = LowLevelCallable(transform(), ptr)
im = np.arange(12).reshape(4, 3).astype(np.float64)
print(ndimage.geometric_transform(im, callback))
```

produces the same result as the original python script.

In the C version `_transform` is the callback function and the parameters `output_coordinates` and `input_coordinates` play the same role as they do in the python version while `output_rank` and `input_rank` provide the equivalents of `len(output_coordinates)` and `len(input_coordinates)`. The variable `shift` is passed through `user_data` instead of `extra_arguments`. Finally, the C callback function returns an integer status which is one upon success and zero otherwise.

The function `py_transform` wraps the callback function in a `PyCapsule`. The main steps are:

- Initialize a `PyCapsule`. The first argument is a pointer to the callback function.

- The second argument is the function signature which must match exactly the one expected by `ndimage`.
- Above, we used *scipy.LowLevelCallable* to specify `user_data` that we generated with `ctypes`.

  A different approach would be to supply the data in the capsule context, that can be set by **:cfunc:‘PyCapsule_SetContext‘** and omit specifying `user_data` in *scipy.LowLevelCallable*. However, in this approach we would need to deal with allocation/freeing of the data — freeing the data after the capsule is destroyed can be done by specifying a non-NULL callback function in the third argument of **:cfunc:‘PyCapsule_New‘**.

C callback functions for `ndimage` all follow this scheme. The next section lists the `ndimage` functions that accept a C callback function and gives the prototype of the function.

See also:

The functions that support low-level callback arguments are:

generic_filter, *generic_filter1d*, *geometric_transform*

Below, we show alternative ways to write the code, using Cython, ctypes, or cffi instead of writing wrapper code in C.

Numba

Numba provides a way to write low-level functions easily in Python. We can write the above using Numba as:

```
# example.py
import numpy as np
import ctypes
from scipy import ndimage, LowLevelCallable
from numba import cfunc, types, carray

@cfunc(types.intc(types.CPointer(types.intp),
                  types.CPointer(types.double),
                  types.intc,
                  types.intc,
                  types.voidptr))
def transform(output_coordinates_ptr, input_coordinates_ptr,
              output_rank, input_rank, user_data):
    input_coordinates = carray(input_coordinates_ptr, (input_rank,))
    output_coordinates = carray(output_coordinates_ptr, (output_rank,))
    shift = carray(user_data, (1,), types.double)[0]

    for i in range(input_rank):
        input_coordinates[i] = output_coordinates[i] - shift

    return 1

shift = 0.5

# Then call the function
user_data = ctypes.c_double(shift)
ptr = ctypes.cast(ctypes.pointer(user_data), ctypes.c_void_p)
callback = LowLevelCallable(transform.ctypes, ptr)

im = np.arange(12).reshape(4, 3).astype(np.float64)
print(ndimage.geometric_transform(im, callback))
```

Cython

Functionally the same code as above can be written in Cython with somewhat less boilerplate as follows.

```
# example.pyx

from numpy cimport npy_intp as intp

cdef api int transform(intp *output_coordinates, double *input_coordinates,
                       int output_rank, int input_rank, void *user_data):
    cdef intp i
    cdef double shift = (<double *>user_data)[0]

    for i in range(input_rank):
        input_coordinates[i] = output_coordinates[i] - shift
    return 1
```

```
# script.py

import ctypes
import numpy as np
from scipy import ndimage, LowLevelCallable

import example

shift = 0.5

user_data = ctypes.c_double(shift)
ptr = ctypes.cast(ctypes.pointer(user_data), ctypes.c_void_p)
callback = LowLevelCallable.from_cython(example, "transform", ptr)
im = np.arange(12).reshape(4, 3).astype(np.float64)
print(ndimage.geometric_transform(im, callback))
```

cffi

With cffi, you can interface with a C function residing in a shared library (DLL). First, we need to write the shared library, which we do in C — this example is for Linux/OSX:

```
/*
  example.c
  Needs to be compiled with "gcc -std=c99 -shared -fPIC -o example.so example.c"
  or similar
 */

#include <stdint.h>

int
_transform(intptr_t *output_coordinates, double *input_coordinates,
           int output_rank, int input_rank, void *user_data)
{
    int i;
    double shift = *(double *)user_data;

    for (i = 0; i < input_rank; i++) {
        input_coordinates[i] = output_coordinates[i] - shift;
    }
    return 1;
}
```

The Python code calling the library is:

```
import os
import numpy as np
from scipy import ndimage, LowLevelCallable
import cffi

# Construct the FFI object, and copypaste the function declaration
ffi = cffi.FFI()
ffi.cdef("""
int _transform(intptr_t *output_coordinates, double *input_coordinates,
               int output_rank, int input_rank, void *user_data);
""")

# Open library
lib = ffi.dlopen(os.path.abspath("example.so"))

# Do the function call
user_data = ffi.new('double *', 0.5)
callback = LowLevelCallable(lib._transform, user_data)
im = np.arange(12).reshape(4, 3).astype(np.float64)
print(ndimage.geometric_transform(im, callback))
```

You can find more information in the cffi documentation.

ctypes

With *ctypes*, the C code and the compilation of the so/DLL is as for cffi above. The Python code is different:

```
# script.py

import os
import ctypes
import numpy as np
from scipy import ndimage, LowLevelCallable

lib = ctypes.CDLL(os.path.abspath('example.so'))

shift = 0.5

user_data = ctypes.c_double(shift)
ptr = ctypes.cast(ctypes.pointer(user_data), ctypes.c_void_p)

# Ctypes has no built-in intptr type, so override the signature
# instead of trying to get it via ctypes
callback = LowLevelCallable(lib._transform, ptr,
    "int _transform(intptr_t *, double *, int, int, void *)")

# Perform the call
im = np.arange(12).reshape(4, 3).astype(np.float64)
print(ndimage.geometric_transform(im, callback))
```

You can find more information in the ctypes documentation.

References

3.1.15 File IO (`scipy.io`)

See also:

numpy-reference.routines.io (in numpy)

MATLAB files

`loadmat`(file_name[, mdict, appendmat])	Load MATLAB file.
`savemat`(file_name, mdict[, appendmat, ...])	Save a dictionary of names and arrays into a MATLAB-style .mat file.
`whosmat`(file_name[, appendmat])	List variables inside a MATLAB file.

The basic functions

We'll start by importing `scipy.io` and calling it `sio` for convenience:

```
>>> import scipy.io as sio
```

If you are using IPython, try tab completing on `sio`. Among the many options, you will find:

```
sio.loadmat
sio.savemat
sio.whosmat
```

These are the high-level functions you will most likely use when working with MATLAB files. You'll also find:

```
sio.matlab
```

This is the package from which `loadmat`, `savemat` and `whosmat` are imported. Within `sio.matlab`, you will find the `mio` module This module contains the machinery that `loadmat` and `savemat` use. From time to time you may find yourself re-using this machinery.

How do I start?

You may have a `.mat` file that you want to read into Scipy. Or, you want to pass some variables from Scipy / Numpy into MATLAB.

To save us using a MATLAB license, let's start in Octave. Octave has MATLAB-compatible save and load functions. Start Octave (`octave` at the command line for me):

```
octave:1> a = 1:12
a =

   1   2   3   4   5   6   7   8   9  10  11  12

octave:2> a = reshape(a, [1 3 4])
a =

ans(:,:,1) =

   1   2   3

ans(:,:,2) =

   4   5   6

ans(:,:,3) =

   7   8   9
```

```
ans(:,:,4) =

   10   11   12

octave:3> save -6 octave_a.mat a % MATLAB 6 compatible
octave:4> ls octave_a.mat
octave_a.mat
```

Now, to Python:

```
>>> mat_contents = sio.loadmat('octave_a.mat')
>>> mat_contents
{'a': array([[[  1.,    4.,    7.,   10.],
        [  2.,    5.,    8.,   11.],
        [  3.,    6.,    9.,   12.]]]),
 '__version__': '1.0',
 '__header__': 'MATLAB 5.0 MAT-file, written by
 Octave 3.6.3, 2013-02-17 21:02:11 UTC',
 '__globals__': []}
>>> oct_a = mat_contents['a']
>>> oct_a
array([[[  1.,    4.,    7.,   10.],
        [  2.,    5.,    8.,   11.],
        [  3.,    6.,    9.,   12.]]])
>>> oct_a.shape
(1, 3, 4)
```

Now let's try the other way round:

```
>>> import numpy as np
>>> vect = np.arange(10)
>>> vect.shape
(10,)
>>> sio.savemat('np_vector.mat', {'vect':vect})
```

Then back to Octave:

```
octave:8> load np_vector.mat
octave:9> vect
vect =

  0  1  2  3  4  5  6  7  8  9

octave:10> size(vect)
ans =

    1   10
```

If you want to inspect the contents of a MATLAB file without reading the data into memory, use the `whosmat` command:

```
>>> sio.whosmat('octave_a.mat')
[('a', (1, 3, 4), 'double')]
```

`whosmat` returns a list of tuples, one for each array (or other object) in the file. Each tuple contains the name, shape and data type of the array.

MATLAB structs

MATLAB structs are a little bit like Python dicts, except the field names must be strings. Any MATLAB object can be a value of a field. As for all objects in MATLAB, structs are in fact arrays of structs, where a single struct is an array of shape (1, 1).

```
octave:11> my_struct = struct('field1', 1, 'field2', 2)
my_struct =
{
  field1 =  1
  field2 =  2
}

octave:12> save -6 octave_struct.mat my_struct
```

We can load this in Python:

```
>>> mat_contents = sio.loadmat('octave_struct.mat')
>>> mat_contents
{'my_struct': array([[([[1.0]], [[2.0]])]],
      dtype=[('field1', 'O'), ('field2', 'O')]), '__version__': '1.0', '__header__':
↪'MATLAB 5.0 MAT-file, written by Octave 3.6.3, 2013-02-17 21:23:14 UTC', '__globals_
↪_': []}
>>> oct_struct = mat_contents['my_struct']
>>> oct_struct.shape
(1, 1)
>>> val = oct_struct[0,0]
>>> val
([[1.0]], [[2.0]])
>>> val['field1']
array([[ 1.]])
>>> val['field2']
array([[ 2.]])
>>> val.dtype
dtype([('field1', 'O'), ('field2', 'O')])
```

In versions of Scipy from 0.12.0, MATLAB structs come back as numpy structured arrays, with fields named for the struct fields. You can see the field names in the `dtype` output above. Note also:

```
>>> val = oct_struct[0,0]
```

and:

```
octave:13> size(my_struct)
ans =

   1   1
```

So, in MATLAB, the struct array must be at least 2D, and we replicate that when we read into Scipy. If you want all length 1 dimensions squeezed out, try this:

```
>>> mat_contents = sio.loadmat('octave_struct.mat', squeeze_me=True)
>>> oct_struct = mat_contents['my_struct']
>>> oct_struct.shape
()
```

Sometimes, it's more convenient to load the MATLAB structs as python objects rather than numpy structured arrays - it can make the access syntax in python a bit more similar to that in MATLAB. In order to do this, use the `struct_as_record=False` parameter setting to `loadmat`.

```
>>> mat_contents = sio.loadmat('octave_struct.mat', struct_as_record=False)
>>> oct_struct = mat_contents['my_struct']
>>> oct_struct[0,0].field1
array([[ 1.]])
```

`struct_as_record=False` works nicely with `squeeze_me`:

```
>>> mat_contents = sio.loadmat('octave_struct.mat', struct_as_record=False, squeeze_
↪me=True)
>>> oct_struct = mat_contents['my_struct']
>>> oct_struct.shape # but no - it's a scalar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'mat_struct' object has no attribute 'shape'
>>> type(oct_struct)
<class 'scipy.io.matlab.mio5_params.mat_struct'>
>>> oct_struct.field1
1.0
```

Saving struct arrays can be done in various ways. One simple method is to use dicts:

```
>>> a_dict = {'field1': 0.5, 'field2': 'a string'}
>>> sio.savemat('saved_struct.mat', {'a_dict': a_dict})
```

loaded as:

```
octave:21> load saved_struct
octave:22> a_dict
a_dict =

  scalar structure containing the fields:

    field2 = a string
    field1 =  0.50000
```

You can also save structs back again to MATLAB (or Octave in our case) like this:

```
>>> dt = [('f1', 'f8'), ('f2', 'S10')]
>>> arr = np.zeros((2,), dtype=dt)
>>> arr
array([(0.0, ''), (0.0, '')],
      dtype=[('f1', '<f8'), ('f2', 'S10')])
>>> arr[0]['f1'] = 0.5
>>> arr[0]['f2'] = 'python'
>>> arr[1]['f1'] = 99
>>> arr[1]['f2'] = 'not perl'
>>> sio.savemat('np_struct_arr.mat', {'arr': arr})
```

MATLAB cell arrays

Cell arrays in MATLAB are rather like python lists, in the sense that the elements in the arrays can contain any type of MATLAB object. In fact they are most similar to numpy object arrays, and that is how we load them into numpy.

```
octave:14> my_cells = {1, [2, 3]}
my_cells =
{
  [1,1] =  1
  [1,2] =
```

```
    2   3

}

octave:15> save -6 octave_cells.mat my_cells
```

Back to Python:

```
>>> mat_contents = sio.loadmat('octave_cells.mat')
>>> oct_cells = mat_contents['my_cells']
>>> print(oct_cells.dtype)
object
>>> val = oct_cells[0,0]
>>> val
array([[ 1.]])
>>> print(val.dtype)
float64
```

Saving to a MATLAB cell array just involves making a numpy object array:

```
>>> obj_arr = np.zeros((2,), dtype=np.object)
>>> obj_arr[0] = 1
>>> obj_arr[1] = 'a string'
>>> obj_arr
array([1, 'a string'], dtype=object)
>>> sio.savemat('np_cells.mat', {'obj_arr':obj_arr})
```

```
octave:16> load np_cells.mat
octave:17> obj_arr
obj_arr =
{
  [1,1] = 1
  [2,1] = a string
}
```

IDL files

readsav(file_name[, idict, python_dict, ...])	Read an IDL .sav file.

Matrix Market files

mminfo(source)	Return size and storage parameters from Matrix Market file-like 'source'.
mmread(source)	Reads the contents of a Matrix Market file-like 'source' into a matrix.
mmwrite(target, a[, comment, field, ...])	Writes the sparse or dense array *a* to Matrix Market file-like *target*.

Wav sound files (`scipy.io.wavfile`)

read(filename[, mmap])	Open a WAV file
write(filename, rate, data)	Write a numpy array as a WAV file.

Arff files (scipy.io.arff)

Module to read ARFF files, which are the standard data format for WEKA.

ARFF is a text file format which support numerical, string and data values. The format can also represent missing data and sparse data.

Notes

The ARFF support in `scipy.io` provides file reading functionality only. For more extensive ARFF functionality, see liac-arff.

See the WEKA website for more details about the ARFF format and available datasets.

loadarff(f)	Read an arff file.

Netcdf (scipy.io.netcdf)

netcdf_file(filename[, mode, mmap, version, ...])	A file object for NetCDF data.

Allows reading of NetCDF files (version of pupynere package)

CHAPTER

FOUR

DEVELOPER'S GUIDE

Explanations of how to start contributing to SciPy, and descriptions of maintenance activities and policies.

4.1 Contributing to SciPy

This document aims to give an overview of how to contribute to SciPy. It tries to answer commonly asked questions, and provide some insight into how the community process works in practice. Readers who are familiar with the SciPy community and are experienced Python coders may want to jump straight to the git workflow documentation.

Note: You may want to check the latest version of this guide, which is available at: https://github.com/scipy/scipy/blob/master/HACKING.rst.txt

4.1.1 Contributing new code

If you have been working with the scientific Python toolstack for a while, you probably have some code lying around of which you think "this could be useful for others too". Perhaps it's a good idea then to contribute it to SciPy or another open source project. The first question to ask is then, where does this code belong? That question is hard to answer here, so we start with a more specific one: *what code is suitable for putting into SciPy?* Almost all of the new code added to scipy has in common that it's potentially useful in multiple scientific domains and it fits in the scope of existing scipy submodules. In principle new submodules can be added too, but this is far less common. For code that is specific to a single application, there may be an existing project that can use the code. Some scikits (scikit-learn, scikit-image, statsmodels, etc.) are good examples here; they have a narrower focus and because of that more domain-specific code than SciPy.

Now if you have code that you would like to see included in SciPy, how do you go about it? After checking that your code can be distributed in SciPy under a compatible license (see FAQ for details), the first step is to discuss on the scipy-dev mailing list. All new features, as well as changes to existing code, are discussed and decided on there. You can, and probably should, already start this discussion before your code is finished.

Assuming the outcome of the discussion on the mailing list is positive and you have a function or piece of code that does what you need it to do, what next? Before code is added to SciPy, it at least has to have good documentation, unit tests and correct code style.

1. ***Unit tests*** In principle you should aim to create unit tests that exercise all the code that you are adding. This gives some degree of confidence that your code runs correctly, also on Python versions and hardware or OSes that you don't have available yourself. An extensive description of how to write unit tests is given in the NumPy testing guidelines.

2. ***Documentation***

   Clear and complete documentation is essential in order for users to be able to find and understand the code. Documentation for individual functions and classes – which includes at least a basic description, type and meaning of all parameters and returns values, and usage examples in doctest format – is put in docstrings. Those docstrings can be read within the interpreter, and are compiled into a reference guide in html and pdf format. Higher-level documentation for key (areas of) functionality is provided in tutorial format and/or in module docstrings. A guide on how to write documentation is given in how to document.

3. ***Code style*** Uniformity of style in which code is written is important to others trying to understand the code. SciPy follows the standard Python guidelines for code style, PEP8. In order to check that your code conforms to PEP8, you can use the pep8 package style checker. Most IDEs and text editors have settings that can help you follow PEP8, for example by translating tabs by four spaces. Using pyflakes to check your code is also a good idea.

At the end of this document a checklist is given that may help to check if your code fulfills all requirements for inclusion in SciPy.

Another question you may have is: *where exactly do I put my code*? To answer this, it is useful to understand how the SciPy public API (application programming interface) is defined. For most modules the API is two levels deep, which means your new function should appear as `scipy.submodule.my_new_func`. `my_new_func` can be put in an existing or new file under `/scipy/<submodule>/`, its name is added to the `__all__` list in that file (which lists all public functions in the file), and those public functions are then imported in `/scipy/<submodule>/__init__.py`. Any private functions/classes should have a leading underscore (_) in their name. A more detailed description of what the public API of SciPy is, is given in SciPy API.

Once you think your code is ready for inclusion in SciPy, you can send a pull request (PR) on Github. We won't go into the details of how to work with git here, this is described well in the git workflow section of the NumPy documentation and on the Github help pages. When you send the PR for a new feature, be sure to also mention this on the scipy-dev mailing list. This can prompt interested people to help review your PR. Assuming that you already got positive feedback before on the general idea of your code/feature, the purpose of the code review is to ensure that the code is correct, efficient and meets the requirements outlined above. In many cases the code review happens relatively quickly, but it's possible that it stalls. If you have addressed all feedback already given, it's perfectly fine to ask on the mailing list again for review (after a reasonable amount of time, say a couple of weeks, has passed). Once the review is completed, the PR is merged into the "master" branch of SciPy.

The above describes the requirements and process for adding code to SciPy. It doesn't yet answer the question though how decisions are made exactly. The basic answer is: decisions are made by consensus, by everyone who chooses to participate in the discussion on the mailing list. This includes developers, other users and yourself. Aiming for consensus in the discussion is important – SciPy is a project by and for the scientific Python community. In those rare cases that agreement cannot be reached, the maintainers of the module in question can decide the issue.

4.1.2 Contributing by helping maintain existing code

The previous section talked specifically about adding new functionality to SciPy. A large part of that discussion also applies to maintenance of existing code. Maintenance means fixing bugs, improving code quality or style, documenting existing functionality better, adding missing unit tests, keeping build scripts up-to-date, etc. The SciPy issue list contains all reported bugs, build/documentation issues, etc. Fixing issues helps improve the overall quality of SciPy, and is also a good way of getting familiar with the project. You may also want to fix a bug because you ran into it and need the function in question to work correctly.

The discussion on code style and unit testing above applies equally to bug fixes. It is usually best to start by writing a unit test that shows the problem, i.e. it should pass but doesn't. Once you have that, you can fix the code so that the test does pass. That should be enough to send a PR for this issue. Unlike when adding new code, discussing this on the mailing list may not be necessary - if the old behavior of the code is clearly incorrect, no one will object to having

it fixed. It may be necessary to add some warning or deprecation message for the changed behavior. This should be part of the review process.

4.1.3 Other ways to contribute

There are many ways to contribute other than contributing code. Participating in discussions on the scipy-user and scipy-dev *mailing lists* is a contribution in itself. The scipy.org *website* contains a lot of information on the SciPy community and can always use a new pair of hands.

4.1.4 Recommended development setup

Since Scipy contains parts written in C, C++, and Fortran that need to be compiled before use, make sure you have the necessary compilers and Python development headers installed. Having compiled code also means that importing Scipy from the development sources needs some additional steps, which are explained below.

First fork a copy of the main Scipy repository in Github onto your own account and then create your local repository via:

```
$ git clone git@github.com:YOURUSERNAME/scipy.git scipy
$ cd scipy
$ git remote add upstream git://github.com/scipy/scipy.git
```

To build the development version of Scipy and run tests, spawn interactive shells with the Python import paths properly set up etc., do one of:

```
$ python runtests.py -v
$ python runtests.py -v -s optimize
$ python runtests.py -v -t scipy/special/tests/test_basic.py:test_xlogy
$ python runtests.py --ipython
$ python runtests.py --python somescript.py
$ python runtests.py --bench
```

This builds Scipy first, so the first time it may take some time. If you specify `-n`, the tests are run against the version of Scipy (if any) found on current PYTHONPATH.

Using `runtests.py` is the recommended approach to running tests. There are also a number of alternatives to it, for example in-place build or installing to a virtualenv. See the FAQ below for details.

Some of the tests in Scipy are very slow and need to be separately enabled. See the FAQ below for details.

4.1.5 SciPy structure

All SciPy modules should follow the following conventions. In the following, a *SciPy module* is defined as a Python package, say `yyy`, that is located in the scipy/ directory.

- Ideally, each SciPy module should be as self-contained as possible. That is, it should have minimal dependencies on other packages or modules. Even dependencies on other SciPy modules should be kept to a minimum. A dependency on NumPy is of course assumed.
- Directory `yyy/` contains:
  - A file `setup.py` that defines `configuration(parent_package='',top_path=None)` function for `numpy.distutils`.
  - A directory `tests/` that contains files `test_<name>.py` corresponding to modules `yyy/<name>{.py,.so,/}`.

- Private modules should be prefixed with an underscore _, for instance `yyy/_somemodule.py`.
- User-visible functions should have good documentation following the Numpy documentation style, see how to document
- The `__init__.py` of the module should contain the main reference documentation in its docstring. This is connected to the Sphinx documentation under `doc/` via Sphinx's automodule directive.

  The reference documentation should first give a categorized list of the contents of the module using `autosummary::` directives, and after that explain points essential for understanding the use of the module.

  Tutorial-style documentation with extensive examples should be separate, and put under `doc/source/tutorial/`

See the existing Scipy submodules for guidance.

For further details on Numpy distutils, see:

https://github.com/numpy/numpy/blob/master/doc/DISTUTILS.rst.txt

4.1.6 Useful links, FAQ, checklist

Checklist before submitting a PR

- Are there unit tests with good code coverage?
- Do all public function have docstrings including examples?
- Is the code style correct (PEP8, pyflakes)
- Is the commit message formatted correctly?
- Is the new functionality tagged with `.. versionadded:: X.Y.Z` (with X.Y.Z the version number of the next release - can be found in setup.py)?
- Is the new functionality mentioned in the release notes of the next release?
- Is the new functionality added to the reference guide?
- In case of larger additions, is there a tutorial or more extensive module-level description?
- In case compiled code is added, is it integrated correctly via setup.py (and preferably also Bento configuration files - bento.info and bscript)?
- If you are a first-time contributor, did you add yourself to THANKS.txt? Please note that this is perfectly normal and desirable - the aim is to give every single contributor credit, and if you don't add yourself it's simply extra work for the reviewer (or worse, the reviewer may forget).
- Did you check that the code can be distributed under a BSD license?

Useful SciPy documents

- The how to document guidelines
- NumPy/SciPy testing guidelines
- SciPy API
- The SciPy Roadmap
- SciPy maintainers
- NumPy/SciPy git workflow

- How to submit a good bug report

FAQ

I based my code on existing Matlab/R/... code I found online, is this OK?

It depends. SciPy is distributed under a BSD license, so if the code that you based your code on is also BSD licensed or has a BSD-compatible license (MIT, Apache, ...) then it's OK. Code which is GPL-licensed, has no clear license, requires citation or is free for academic use only can't be included in SciPy. Therefore if you copied existing code with such a license or made a direct translation to Python of it, your code can't be included. See also license compatibility.

Why is SciPy under the BSD license and not, say, the GPL?

Like Python, SciPy uses a "permissive" open source license, which allows proprietary re-use. While this allows companies to use and modify the software without giving anything back, it is felt that the larger user base results in more contributions overall, and companies often publish their modifications anyway, without being required to. See John Hunter's BSD pitch.

How do I set up a development version of SciPy in parallel to a released version that I use to do my job/research?

One simple way to achieve this is to install the released version in site-packages, by using a binary installer or pip for example, and set up the development version in a virtualenv. First install virtualenv (optionally use virtualenvwrapper), then create your virtualenv (named scipy-dev here) with:

```
$ virtualenv scipy-dev
```

Now, whenever you want to switch to the virtual environment, you can use the command `source scipy-dev/bin/activate`, and `deactivate` to exit from the virtual environment and back to your previous shell. With scipy-dev activated, install first Scipy's dependencies:

```
$ pip install Numpy Nose Cython
```

After that, you can install a development version of Scipy, for example via:

```
$ python setup.py install
```

The installation goes to the virtual environment.

How do I set up an in-place build for development

For development, you can set up an in-place build so that changes made to `.py` files have effect without rebuild. First, run:

```
$ python setup.py build_ext -i
```

Then you need to point your PYTHONPATH environment variable to this directory. Some IDEs (Spyder for example) have utilities to manage PYTHONPATH. On Linux and OSX, you can run the command:

```
$ export PYTHONPATH=$PWD
```

and on Windows

$ set PYTHONPATH=/path/to/scipy

Now editing a Python source file in SciPy allows you to immediately test and use your changes (in `.py` files), by simply restarting the interpreter.

Can I use a programming language other than Python to speed up my code?

Yes. The languages used in SciPy are Python, Cython, C, C++ and Fortran. All of these have their pros and cons. If Python really doesn't offer enough performance, one of those languages can be used. Important concerns when using compiled languages are maintainability and portability. For maintainability, Cython is clearly preferred over C/C++/Fortran. Cython and C are more portable than C++/Fortran. A lot of the existing C and Fortran code in SciPy is older, battle-tested code that was only wrapped in (but not specifically written for) Python/SciPy. Therefore the basic advice is: use Cython. If there's specific reasons why C/C++/Fortran should be preferred, please discuss those reasons first.

How do I debug code written in C/C++/Fortran inside Scipy?

The easiest way to do this is to first write a Python script that invokes the C code whose execution you want to debug. For instance `mytest.py`:

```
from scipy.special import hyp2f1
print(hyp2f1(5.0, 1.0, -1.8, 0.95))
```

Now, you can run:

```
gdb --args python runtests.py -g --python mytest.py
```

If you didn't compile with debug symbols enabled before, remove the `build` directory first. While in the debugger:

```
(gdb) break cephes_hyp2f1
(gdb) run
```

The execution will now stop at the corresponding C function and you can step through it as usual. Instead of plain `gdb` you can of course use your favourite alternative debugger; run it on the `python` binary with arguments `runtests.py -g --python mytest.py`.

How do I enable additional tests in Scipy?

Some of the tests in Scipy's test suite are very slow and not enabled by default. You can run the full suite via:

```
$ python runtests.py -g -m full
```

This invokes the test suite `import scipy; scipy.test("full")`, enabling also slow tests.

There is an additional level of very slow tests (several minutes), which are disabled also in this case. They can be enabled by setting the environment variable `SCIPY_XSLOW=1` before running the test suite.

How do I write tests with test generators?

The Nose test framework supports so-called test generators, which can come useful if you need to have multiple tests where just a parameter changes. Using test generators so that they are more useful than harmful is tricky, and we recommend the following pattern:

```
def test_something():
    some_array = (...)

    def check(some_param):
        c = compute_result(some_array, some_param)
        known_result = (...)
        assert_allclose(c, known_result)

    for some_param in ['a', 'b', 'c']:
        yield check, some_param
```

We require the following:

- All asserts and all computation that is tested must only be reached after a yield. (Rationale: the generator body is part of no test, and a failure in it will show neither the test name nor for what parameters the test failed.)
- Arrays must not be passed as yield parameters. Either use variables from outer scope (eg. with some index passed to yield), or capsulate test data to a class with a sensible `__repr__`. (Rationale: Nose truncates the printed form of arrays in test output, and this makes it impossible to know for what parameters a test failed. Arrays are big, and clutter test output unnecessarily.)
- Test generators cannot be used in test classes inheriting from unittest.TestCase; either use object as base class, or use standalone test functions. (Rationale: Nose does not run test generators in TestCase-inheriting classes.)

If in doubt, do not use test generators. You can track for what parameter things failed also by passing `err_msg=repr((param1, param2, ...))` to the various assert functions.

4.2 SciPy Developer Guide

4.2.1 Decision making process

This section documents the way in which decisions about various aspects of the SciPy project are made. Note that the below is only documenting the current way of working; a more formal governance model is expected to be adopted in the near future.

Code

Any significant decisions on adding (or not adding) new features, breaking backwards compatibility or making other significant changes to the codebase should be made on the scipy-dev mailing list after a discussion (preferably with full consensus).

Any non-trivial change (where trivial means a typo, or a one-liner maintenance commit) has to go in through a pull request (PR). It has to be reviewed by another developer. In case review doesn't happen quickly enough and it is important that the PR is merged quickly, the submitter of the PR should send a message to mailing list saying he/she intends to merge that PR without review at time X for reason Y unless someone reviews it before then.

Changes and new additions should be tested. Untested code is broken code.

Commit rights

Who gets commit rights is decided by the core development team; changes in commit rights will then be announced on the scipy-dev mailing list.

Who the core development team is comprised of is a little fuzzy - there are quite a few people who do have commit rights and would like to keep them but are no longer active (so they're not in the core team). To get an idea, look at the output of:

```
$ git shortlog --grep="Merge pull request" -a -c -s <current_release_minus_2>..
↪upstream/master|sort -n
```

and apply some common sense to it (and don't forget people who are still active but never merge PRs).

Other project aspects

All decisions are taken by the core development team on the scipy-dev mailing list.

4.2.2 Deciding on new features

The general decision rule to accept a proposed new feature has so far been conditional on:

1. The method is applicable in many fields and "generally agreed" to be useful,
2. Fits the topic of the submodule, and does not require extensive support frameworks to operate,
3. The implementation looks sound and unlikely to need much tweaking in the future (e.g., limited expected maintenance burden), and
4. Someone wants to do it.

Although it's difficult to give hard rules on what "generally useful and generally agreed to work" means, it may help to weigh the following against each other:

- Is the method used/useful in different domains in practice? How much domain-specific background knowledge is needed to use it properly?
- Consider the code already in the module. Is what you are adding an omission? Does it solve a problem that you'd expect the module be able to solve? Does it supplement an existing feature in a significant way?
- Consider the equivalence class of similar methods / features usually expected. Among them, what would in principle be the minimal set so that there's not a glaring omission in the offered features remaining? How much stuff would that be? Does including a representative one of them cover most use cases? Would it in principle sound reasonable to include everything from the minimal set in the module?
- Is what you are adding something that is well understood in the literature? If not, how sure are you that it will turn out well? Does the method perform well compared to other similar ones?
- Note that the twice-a-year release cycle and backward-compatibility policy makes correcting things later on more difficult.

The scopes of the submodules also vary, so it's probably best to consider each as if it's a separate project - "numerical evaluation of special functions" is relatively well-defined, but "commonly needed optimization algorithms" less so.

4.2.3 Development on GitHub

SciPy development largely takes place on GitHub; this section describes the expected way of working for issues, pull requests and managing the main `scipy` repository.

Labels and Milestones

Each issue and pull request normally gets at least two labels: one for the topic or component (`scipy.stats`, `Documentation`, etc.), and one for the nature of the issue or pull request (`enhancement`, `maintenance`, `defect`, etc.). Other labels that may be added depending on the situation:

- `easy-fix`: for issues suitable to be tackled by new contributors.
- `needs-work`: for pull requests that have review comments that haven't been addressed for a while.
- `needs-decision`: for issues or pull requests that need a decision.
- `needs-champion`: for pull requests that were not finished by the original author, but are worth resurrecting.
- `backport-candidate`: bugfixes that should be considered for backporting by the release manager.

A milestone is created for each version number for which a release is planned. Issues that need to be addressed and pull requests that need to be merged for a particular release should be set to the corresponding milestone. After a pull request is merged, its milestone (and that of the issue it closes) should be set to the next upcoming release - this makes it easy to get an overview of changes and to add a complete list of those to the release notes.

Dealing with pull requests

- When merging contributions, a committer is responsible for ensuring that those meet the requirements outlined in Contributing to SciPy. Also check that new features and backwards compatibility breaks were discussed on the scipy-dev mailing list.
- New code goes in via a pull request (PR).
- Merge new code with the green button. In case of merge conflicts, ask the PR submitter to rebase (this may require providing some git instructions).
- Backports and trivial additions to finish a PR (really trivial, like a typo or PEP8 fix) can be pushed directly.
- For PRs that add new features or are in some way complex, wait at least a day or two before merging it. That way, others get a chance to comment before the code goes in.
- Squashing commits or cleaning up commit messages of a PR that you consider too messy is OK. Make sure though to retain the original author name when doing this.
- Make sure that the labels and milestone on a merged PR are set correctly.
- When you want to reject a PR: if it's very obvious you can just close it and explain why, if not obvious then it's a good idea to first explain why you think the PR is not suitable for inclusion in Scipy and then let a second committer comment or close.

Backporting

All pull requests (whether they contain enhancements, bug fixes or something else), should be made against master. Only bug fixes are candidates for backporting to a maintenance branch. The backport strategy for SciPy is to (a) only backport fixes that are important, and (b) to only backport when it's reasonably sure that a new bugfix release on the relevant maintenance branch will be made. Typically, the developer who merges an important bugfix adds the `backport-candidate` label and pings the release manager, who decides on whether and when the backport is done. After the backport is completed, the `backport-candidate` label has to be removed again.

Other

PR status page: When new commits get added to a pull request, GitHub doesn't send out any notifications. The `needs-work` label may not be justified anymore though. This page gives an overview of PRs that were updated, need review, need a decision, etc.

Cross-referencing: Cross-referencing issues and pull requests on GitHub is often useful. GitHub allows doing that by using `gh-xxxx` or `#xxxx` with `xxxx` the issue/PR number. The `gh-xxxx` format is strongly preferred, because it's clear that that is a GitHub link. Older issues contain `#xxxx` which is about Trac (what we used pre-GitHub) tickets.

PR naming convention: Pull requests, issues and commit messages usually start with a three-letter abbreviation like `ENH:` or `BUG:`. This is useful to quickly see what the nature of the commit/PR/issue is. For the full list of abbreviations, see writing the commit message.

4.2.4 Licensing

Scipy is distributed under the modified (3-clause) BSD license. All code, documentation and other files added to Scipy by contributors is licensed under this license, unless another license is explicitly specified in the source code. Contributors keep the copyright for code they wrote and submit for inclusion to Scipy.

Other licenses that are compatible with the modified BSD license that Scipy uses are 2-clause BSD, MIT and PSF. Incompatible licenses are GPL, Apache and custom licenses that require attribution/citation or prohibit use for commercial purposes.

It regularly happens that PRs are submitted with content copied or derived from unlicensed code. Such contributions cannot be accepted for inclusion in Scipy. What is needed in such cases is to contact the original author and ask him to relicense his code under the modified BSD (or a compatible) license. If the original author agrees to this, add a comment saying so to the source files and forward the relevant email to the scipy-dev mailing list.

What also regularly happens is that code is translated or derived from code in R, Octave (both GPL-licensed) or a commercial application. Such code also cannot be included in Scipy. Simply implementing functionality with the same API as found in R/Octave/... is fine though, as long as the author doesn't look at the original incompatibly-licensed source code.

4.2.5 Version numbering

Scipy version numbering complies to PEP 440. Released final versions, which are the only versions appearing on PyPI, are numbered `MAJOR.MINOR.MICRO` where:

- `MAJOR` is an integer indicating the major version. It changes very rarely; a change in `MAJOR` indicates large (possibly backwards-incompatible) changes.
- `MINOR` is an integer indicating the minor version. Minor versions are typically released twice a year and can contain new features, deprecations and bug-fixes.
- `MICRO` is an integer indicating a bug-fix version. Bug-fix versions are released when needed, typically one or two per minor version. They cannot contain new features or deprecations.

Released alpha, beta and rc (release candidate) versions are numbered like final versions but with postfixes `a#`, `b#` and `rc#` respectively, with `#` an integer. Development versions are postfixed with `.dev0+<git-commit-hash>`.

Examples of valid Scipy version strings are:

```
0.16.0
0.15.1
0.14.0a1
0.14.0b2
0.14.0rc1
0.17.0.dev0+ac53f09
```

An installed Scipy version contains these version identifiers:

```
scipy.__version__            # complete version string, including git commit hash for␣
↪dev versions
scipy.version.short_version  # string, only major.minor.micro
scipy.version.version        # string, same as scipy.__version__
scipy.version.full_version   # string, same as scipy.__version__
scipy.version.release        # bool, development or (alpha/beta/rc/final) released␣
↪version
scipy.version.git_revision   # string, git commit hash from which scipy was built
```

4.2.6 Deprecations

There are various reasons for wanting to remove existing functionality: it's buggy, the API isn't understandable, it's superceded by functionality with better performance, it needs to be moved to another Scipy submodule, etc.

In general it's not a good idea to remove something without warning users about that removal first. Therefore this is what should be done before removing something from the public API:

1. Propose to deprecate the functionality on the scipy-dev mailing list and get agreement that that's OK.
2. Add a `DeprecationWarning` for it, which states that the functionality was deprecated, and in which release.

3. Mention the deprecation in the release notes for that release.
4. Wait till at least 6 months after the release date of the release that introduced the `DeprecationWarning` before removing the functionality.
5. Mention the removal of the functionality in the release notes.

The 6 months waiting period in practice usually means waiting two releases. When introducing the warning, also ensure that those warnings are filtered out when running the test suite so they don't pollute the output.

It's possible that there is reason to want to ignore this deprecation policy for a particular deprecation; this can always be discussed on the scipy-dev mailing list.

4.2.7 Distributing

Distributing Python packages is nontrivial - especially for a package with complex build requirements like Scipy - and subject to change. For an up-to-date overview of recommended tools and techniques, see the Python Packaging User Guide. This document discusses some of the main issues and considerations for Scipy.

Dependencies

Dependencies are things that a user has to install in order to use (or build/test) a package. They usually cause trouble, especially if they're not optional. Scipy tries to keep its dependencies to a minimum; currently they are:

Unconditional run-time dependencies:

- Numpy

Conditional run-time dependencies:

- nose (to run the test suite)
- asv (to run the benchmarks)
- matplotlib (for some functions that can produce plots)
- Pillow (for image loading/saving)
- scikits.umfpack (optionally used in `sparse.linalg`)
- mpmath (for more extended tests in `special`)

Unconditional build-time dependencies:

- Numpy
- A BLAS and LAPACK implementation (reference BLAS/LAPACK, ATLAS, OpenBLAS, MKL, Accelerate are all known to work)
- (for development versions) Cython

Conditional build-time dependencies:

- setuptools
- wheel (`python setup.py bdist_wheel`)
- Sphinx (docs)
- matplotlib (docs)
- LaTeX (pdf docs)
- Pillow (docs)

Furthermore of course one needs C, C++ and Fortran compilers to build Scipy, but those we don't consider to be dependencies and are therefore not discussed here. For details, see http://scipy.org/scipylib/building/index.html.

When a package provides useful functionality and it's proposed as a new dependency, consider also if it makes sense to vendor (i.e. ship a copy of it with scipy) the package instead. For example, six and decorator are vendored in `scipy._lib`.

The only dependency that is reported to pip is Numpy, see `install_requires` in Scipy's main `setup.py`. The other dependencies aren't needed for Scipy to function correctly, and the one unconditional build dependency that pip knows how to install (Cython) we prefer to treat like a compiler rather than a Python package that pip is allowed to upgrade.

Issues with dependency handling

There are some serious issues with how Python packaging tools handle dependencies reported by projects. Because Scipy gets regular bug reports about this, we go in a bit of detail here.

Scipy only reports its dependency on Numpy via `install_requires` if Numpy isn't installed at all on a system. This will only change when there are either 32-bit and 64-bit Windows wheels for Numpy on PyPI or when `pip upgrade` becomes available (with sane behavior, unlike `pip install -U`, see this PR). For more details, see this summary.

The situation with `setup_requires` is even worse; pip doesn't handle that keyword at all, while `setuptools` has issues (here's a current one) and invokes `easy_install` which comes with its own set of problems (note that Scipy doesn't support `easy_install` at all anymore; issues specific to it will be closed as "wontfix").

Supported Python and Numpy versions

The Python versions that Scipy supports are listed in the list of PyPI classifiers in `setup.py`, and mentioned in the release notes for each release. All newly released Python versions will be supported as soon as possible. The general policy on dropping support for a Python version is that (a) usage of that version has to be quite low (say <5% of users) and (b) the version isn't included in an active long-term support release of one of the main Linux distributions anymore. Scipy typically follows Numpy, which has a similar policy. The final decision on dropping support is always taken on the scipy-dev mailing list.

The lowest supported Numpy version for a Scipy version is mentioned in the release notes and is encoded in `scipy/__init__.py` and the `install_requires` field of `setup.py`. Typically the latest Scipy release supports 3 or 4 minor versions of Numpy. That may become more if the frequency of Numpy releases increases (it's about 1x/year at the time of writing). Support for a particular Numpy version is typically dropped if (a) that Numpy version is several years old, and (b) the maintenance cost of keeping support is starting to outweigh the benefits. The final decision on dropping support is always taken on the scipy-dev mailing list.

Supported versions of optional dependencies and compilers is less clearly documented, and also isn't tested well or at all by Scipy's Continuous Integration setup. Issues regarding this are dealt with as they come up in the issue tracker or mailing list.

Building binary installers

Note: This section is only about building Scipy binary installers to *distribute*. For info on building Scipy on the same machine as where it will be used, see this scipy.org page.

There are a number of things to take into consideration when building binaries and distributing them on PyPI or elsewhere.

General

- A binary is specific for a single Python version (because different Python versions aren't ABI-compatible, at least up to Python 3.4).
- Build against the lowest Numpy version that you need to support, then it will work for all Numpy versions with the same major version number (Numpy does maintain backwards ABI compatibility).

Windows

- For 64-bit Windows installers built with a free toolchain, use the method documented at https://github.com/numpy/numpy/wiki/Mingw-static-toolchain. That method will likely be used for Scipy itself once it's clear that the maintenance of that toolchain is sustainable long-term. See the MingwPy project and this thread for details.
- The other way to produce 64-bit Windows installers is with `icc`, `ifort` plus `MKL` (or `MSVC` instead of `icc`). For Intel toolchain instructions see this article and for (partial) MSVC instructions see this wiki page.
- Older Scipy releases contained a .exe "superpack" installer. Those contain 3 complete builds (no SSE, SSE2, SSE3), and were built with https://github.com/numpy/numpy-vendor. That build setup is known to not work well anymore and is no longer supported. It used g77 instead of gfortran, due to complex DLL distribution issues (see gh-2829). Because the toolchain is no longer supported, g77 support isn't needed anymore and Scipy can now include Fortran 90/95 code.

OS X

- To produce OS X wheels that work with various Python versions (from python.org, Homebrew, MacPython), use the build method provided by https://github.com/MacPython/scipy-wheels.
- DMG installers for the Python from python.org on OS X can still be produced by `tools/scipy-macosx-installer/`. Scipy doesn't distribute those installers anymore though, now that there are binary wheels on PyPi.

Linux

Besides PyPi not allowing Linux wheels (which is about to change with PEP 513), there are no specific issues with building binaries. To build a set of wheels for a Linux distribution and providing them in a Wheelhouse, look at the wheel and Wheelhouse docs. A Wheelhouse for wheels compatible with TravisCI is http://wheels.scipy.org.

4.2.8 Making a SciPy release

At the highest level, this is what the release manager does to release a new Scipy version:

1. Propose a release schedule on the scipy-dev mailing list.
2. Create the maintenance branch for the release.
3. Tag the release.
4. Build all release artifacts (sources, installers, docs).
5. Upload the release artifacts.
6. Announce the release.
7. Port relevant changes to release notes and build scripts to master.

In this guide we attempt to describe in detail how to perform each of the above steps. In addition to those steps, which have to be performed by the release manager, here are descriptions of release-related activities and conventions of interest:

- *Backporting*
- *Labels and Milestones*
- versioning

- *Supported Python and Numpy versions*
- deprecations

Proposing a release schedule

A typical release cycle looks like:

- Create the maintenance branch
- Release a beta version
- Release a "release candidate" (RC)
- If needed, release one or more new RCs
- Release the final version once there are no issues with the last release candidate

There's usually at least one week between each of the above steps. Experience shows that a cycle takes between 4 and 8 weeks for a new minor version. Bug-fix versions don't need a beta or RC, and can be done much quicker.

Ideally the final release is identical to the last RC, however there may be minor difference - it's up to the release manager to judge the risk of that. Typically, if compiled code or complex pure Python code changes then a new RC is needed, while a simple bug-fix that's backported from master doesn't require a new RC.

To propose a schedule, send a list with estimated dates for branching and beta/rc/final releases to scipy-dev. In the same email, ask everyone to check if there are important issues/PRs that need to be included and aren't tagged with the Milestone for the release or the "backport-candidate" label.

Creating the maintenance branch

Before branching, ensure that the release notes are updated as far as possible. Include the output of `tools/gh_lists.py` and `tools/authors.py` in the release notes.

Maintenance branches are named `maintenance/0.<minor-version>.x`. To create one, simply push a branch with the correct name to the scipy repo. Immediately after, push a commit where you increment the version number on the master branch and add release notes for that new version. Send an email to scipy-dev to let people know that you've done this.

Tagging a release

First ensure that you have set up GPG correctly. See https://github.com/scipy/scipy/issues/4919 for a discussion of signing release tags, and http://keyring.debian.org/creating-key.html for instructions on creating a GPG key if you do not have one.

To make your key more readily identifiable as you, consider sending your key to public keyservers, with a command such as:

```
gpg --send-keys <yourkeyid>
```

Check that all relevant commits are in the branch. In particular, check issues and PRs under the Milestone for the release (https://github.com/scipy/scipy/milestones), PRs labeled "backport-candidate", and that the release notes are up-to-date and included in the html docs.

Then edit `setup.py` to get the correct version number (set `ISRELEASED = True`) and commit it with a message like `REL: set version to <version-number>`. Don't push this commit to the Scipy repo yet.

Finally tag the release locally with `git tag -s <v0.x.y>` (the `-s` ensures the tag is signed). Continue with building release artifacts (next section). Only push the release commit and tag to the scipy repo once you have built

the docs and Windows installers successfully. After that push, also push a second commit which increment the version number and sets `ISRELEASED` to False again.

Building release artifacts

Here is a complete list of artifacts created for a release:

- source archives (`.tar.gz`, `.zip` and `.tar.xz`)
- Binary wheels, for OS X only at the moment
- Documentation (`html`, `pdf`)
- A `README` file
- A `Changelog` file

All of these except the OS X wheel are built by running `paver release` in the repo root. Do this after you've created the signed tag. If this completes without issues, push the release tag to the scipy repo. This is needed because the OS X wheel build automatically builds the last tag.

To build wheels for OS X, push a commit to the master branch of https://github.com/MacPython/scipy-wheels. This triggers builds for all needed Python versions on TravisCI. Check in the `.travis.yml` config file what version of Python and Numpy are used for the builds (it needs to be the lowest supported Numpy version for each Python version). See the README file in the scipy-wheels repo for more details.

The TravisCI builds run the tests from the built wheels and if they pass upload the wheels to http://wheels.scipy.org/. From there you can copy them for uploading to PyPI (see next section).

Uploading release artifacts

For a release there are currently five places on the web to upload things to:

- PyPI (tarballs, OS X wheels)
- Github releases (tarballs, release notes, Changelog)
- scipy.org (an announcement of the release)
- docs.scipy.org (html/pdf docs)

PyPI:

```
twine upload -s <tarballs or wheels to upload>
```

Github Releases:

Use GUI on https://github.com/scipy/scipy/releases to create release and upload all release artifacts.

SourceForge:

The main download sites are PyPI and Github Releases. Older releases are stored on SourceForce (http://sourceforge.net/projects/scipy/files/scipy). That download site has a "Latest" folder which redirects users to PyPI/GitHub, so it's not needed to upload anything to SourceForge for new releases.

scipy.org:

Sources for the site are in https://github.com/scipy/scipy.org. Update the News section in `www/index.rst` and then do `make upload USERNAME=yourusername`.

docs.scipy.org:

First build the scipy docs, by running `make dist` in `scipy/doc/`. Verify that they look OK, then upload them to the doc server with `make upload USERNAME=rgommers RELEASE=0.17.0`. Note that SSH access to the doc server is needed; ask @pv (server admin) or @rgommers (can upload) if you don't have that.

The sources for the website itself are maintained in https://github.com/scipy/docs.scipy.org/. Add the new Scipy version in the table of releases in `index.rst`. Push that commit, then do `make upload USERNAME=yourusername`.

Wrapping up

Send an email announcing the release to the following mailing lists:

- scipy-dev
- scipy-user
- numpy-discussion
- python-announce (not for beta/rc releases)

For beta and rc versions, ask people in the email to test (run the scipy tests and test against their own code) and report issues on Github or scipy-dev.

After the final release is done, port relevant changes to release notes, build scripts, author name mapping in `tools/authors.py` and any other changes that were only made on the maintenance branch to master.

To get an overview of where help or new features are desired or planned, see the roadmap:

4.3 SciPy Roadmap

Most of this roadmap is intended to provide a high-level view on what is most needed per SciPy submodule in terms of new functionality, bug fixes, etc. Part of those are must-haves for the `1.0` version of Scipy. Furthermore it contains ideas for major new features - those are marked as such, and are not needed for SciPy to become 1.0. Things not mentioned in this roadmap are not necessarily unimportant or out of scope, however we (the SciPy developers) want to provide to our users and contributors a clear picture of where SciPy is going and where help is needed most urgently.

When a module is in a 1.0-ready state, it means that it has the functionality we consider essential and has an API and code quality (including documentation and tests) that's of high enough quality.

4.3.1 General

This roadmap will be evolving together with SciPy. Updates can be submitted as pull requests. For large or disruptive changes you may want to discuss those first on the scipy-dev mailing list.

API changes

In general, we want to take advantage of the major version change to fix some known warts in the API. The change from 0.x.x to 1.x.x is the chance to fix those API issues that we all know are ugly warts. Example: unify the convention for specifying tolerances (including absolute, relative, argument and function value tolerances) of the optimization functions. More API issues will be noted in the module sections below. However, there should be *clear value* in making a breaking change. The 1.0 version label is not a license to just break things - see it as a normal release with a somewhat more aggressive/extensive set of cleanups.

It should be made more clear what is public and what is private in SciPy. Everything private should be underscored as much as possible. Now this is done consistently when we add new code, but for 1.0 it should also be done for existing code.

Test coverage

Test coverage of code added in the last few years is quite good, and we aim for a high coverage for all new code that is added. However, there is still a significant amount of old code for which coverage is poor. Bringing that up to the current standard is probably not realistic, but we should plug the biggest holes. Additionally the coverage should be tracked over time and we should ensure it only goes up.

Besides coverage there is also the issue of correctness - older code may have a few tests that provide decent statement coverage, but that doesn't necessarily say much about whether the code does what it says on the box. Therefore code review of some parts of the code (`stats` and `signal` in particular) is necessary.

Documentation

The documentation is in decent shape. Expanding of current docstrings and putting them in the standard NumPy format should continue, so the number of reST errors and glitches in the html docs decreases. Most modules also have a tutorial in the reference guide that is a good introduction, however there are a few missing or incomplete tutorials - this should be fixed.

Other

Scipy 1.0 will likely contain more backwards-incompatible changes than a minor release. Therefore we will have a longer-lived maintenance branch of the last 0.X release.

Regarding Cython code:

- It's not clear how much functionality can be Cythonized without making the .so files too large. This needs measuring.
- Cython's old syntax for using NumPy arrays should be removed and replaced with Cython memoryviews.
- New feature idea: more of the currently wrapped libraries should export Cython-importable versions that can be used without linking.

Regarding build environments:

- NumPy and SciPy should both build from source on Windows with a MinGW-w64 toolchain and be compatible with Python installations compiled with either the same MinGW or with MSVC.
- Bento development has stopped, so will remain having an experimental, use-at-your-own-risk status. Only the people that use it will be responsible for keeping the Bento build updated.

A more complete continuous integration setup is needed; at the moment we often find out right before a release that there are issues on some less-often used platform or Python version. At least needed are Windows (MSVC and MingwPy), Linux and OS X builds, coverage of the lowest and highest Python and NumPy versions that are supported.

4.3.2 Modules

cluster

This module is in good shape.

constants

This module is basically done, low-maintenance and without open issues.

fftpack

Needed:

- solve issues with single precision: large errors, disabled for difficult sizes
- fix caching bug
- Bluestein algorithm (or chirp Z-transform)
- deprecate fftpack.convolve as public function (was not meant to be public)

There's a large overlap with `numpy.fft`. This duplication has to change (both are too widely used to deprecate one); in the documentation we should make clear that `scipy.fftpack` is preferred over `numpy.fft`. If there are differences in signature or functionality, the best version should be picked case by case (example: numpy's `rfft` is preferred, see gh-2487).

integrate

Needed for ODE solvers:

- Documentation is pretty bad, needs fixing
- A promising new ODE solver interface is in progress: gh-6326. This needs to be finished and merged. After that, older API can possibly be deprecated.

The numerical integration functions are in good shape. Support for integrating complex-valued functions and integrating multiple intervals (see gh-3325) could be added, but is not required for SciPy 1.0.

interpolate

Needed:

- Both fitpack and fitpack2 interfaces will be kept.
- splmake is deprecated; is different spline representation, we need exactly one
- interp1d/interp2d are somewhat ugly but widely used, so we keep them.

Ideas for new features:

- Spline fitting routines with better user control.
- Integration and differentiation and arithmetic routines for splines
- Transparent tensor-product splines.
- NURBS support.
- Mesh refinement and coarsening of B-splines and corresponding tensor products.

io

wavfile;

- PCM float will be supported, for anything else use `audiolab` or other specialized libraries.
- Raise errors instead of warnings if data not understood.

Other sub-modules (matlab, netcdf, idl, harwell-boeing, arff, matrix market) are in good shape.

linalg

Needed:

- Remove functions that are duplicate with `numpy.linalg`
- `get_lapack_funcs` should always use `flapack`
- Wrap more LAPACK functions
- One too many funcs for LU decomposition, remove one

Ideas for new features:

- Add type-generic wrappers in the Cython BLAS and LAPACK
- Make many of the linear algebra routines into gufuncs

misc

`scipy.misc` will be removed as a public module. The functions in it can be moved to other modules:

- `pilutil`, images : `ndimage`
- `comb`, `factorial`, `logsumexp`, `pade` : `special`
- `doccer` : move to `scipy._lib`
- `info`, `who` : these are NumPy functions
- `derivative`, `central_diff_weight` : remove, replace with more extensive functionality for numerical differentiation - likely in a new module `scipy.diff` (see below)

ndimage

Underlying `ndimage` is a powerful interpolation engine. Unfortunately, it was never decided whether to use a pixel model (`(1, 1)` elements with centers `(0.5, 0.5)`) or a data point model (values at points on a grid). Over time, it seems that the data point model is better defined and easier to implement. We therefore propose to move to this data representation for 1.0, and to vet all interpolation code to ensure that boundary values, transformations, etc. are correctly computed. Addressing this issue will close several issues, including #1323, #1903, #2045 and #2640.

The morphology interface needs to be standardized:

- binary dilation/erosion/opening/closing take a "structure" argument, whereas their grey equivalent take size (has to be a tuple, not a scalar), footprint, or structure.
- a scalar should be acceptable for size, equivalent to providing that same value for each axis.
- for binary dilation/erosion/opening/closing, the structuring element is optional, whereas it's mandatory for grey. Grey morphology operations should get the same default.
- other filters should also take that default value where possible.

odr

Rename the module to `regression` or `fitting`, include `optimize.curve_fit`. This module will then provide a home for other fitting functionality - what exactly needs to be worked out in more detail, a discussion can be found at https://github.com/scipy/scipy/pull/448.

optimize

Overall this module is in reasonably good shape, however it is missing a few more good global optimizers as well as large-scale optimizers. These should be added. Other things that are needed:

- deprecate the `fmin_*` functions in the documentation, `minimize` is preferred.
- clearly define what's out of scope for this module.

signal

Convolution and correlation: (Relevant functions are convolve, correlate, fftconvolve, convolve2d, correlate2d, and sepfir2d.) Eliminate the overlap with *ndimage* (and elsewhere). From `numpy`, `scipy.signal` and `scipy.ndimage` (and anywhere else we find them), pick the "best of class" for 1-D, 2-D and n-d convolution and correlation, put the implementation somewhere, and use that consistently throughout SciPy.

B-splines: (Relevant functions are bspline, cubic, quadratic, gauss_spline, cspline1d, qspline1d, cspline2d, qspline2d, cspline1d_eval, and spline_filter.) Move the good stuff to *interpolate* (with appropriate API changes to match how things are done in *interpolate*), and eliminate any duplication.

Filter design: merge *firwin* and *firwin2* so *firwin2* can be removed.

Continuous-Time Linear Systems: remove *lsim2*, *impulse2*, *step2*. Make *lsim*, *impulse* and *step* "just work" for any input system. Improve performance of ltisys (less internal transformations between different representations). Fill gaps in lti system conversion functions.

Second Order Sections: Make SOS filtering equally capable as existing methods. This includes ltisys objects, an *lfiltic* equivalent, and numerically stable conversions to and from other filter representations. SOS filters could be considered as the default filtering method for ltisys objects, for their numerical stability.

Wavelets: what's there now doesn't make much sense. Continous wavelets only at the moment - decide whether to completely rewrite or remove them. Discrete wavelet transforms are out of scope (PyWavelets does a good job for those).

sparse

The sparse matrix formats are getting feature-complete but are slow ... reimplement parts in Cython?

- Small matrices are slower than PySparse, needs fixing

There are a lot of formats. These should be kept, but improvements/optimizations should go into CSR/CSC, which are the preferred formats. LIL may be the exception, it's inherently inefficient. It could be dropped if DOK is extended to support all the operations LIL currently provides. Alternatives are being worked on, see https://github.com/ev-br/sparr and https://github.com/perimosocordiae/sparray.

Ideas for new features:

- Sparse arrays now act like np.matrix. We want sparse *arrays*.

sparse.csgraph

This module is in good shape.

sparse.linalg

Arpack is in good shape.

isolve:

- callback keyword is inconsistent
- tol keyword is broken, should be relative tol
- Fortran code not re-entrant (but we don't solve, maybe re-use from PyKrilov)

dsolve:

- add sparse Cholesky or incomplete Cholesky
- look at CHOLMOD

Ideas for new features:

- Wrappers for PROPACK for faster sparse SVD computation.

spatial

QHull wrappers are in good shape.

Needed:

- `KDTree` will be removed, and `cKDTree` will be renamed to `KDTree` in a backwards-compatible way.
- `distance_wrap.c` needs to be cleaned up (maybe rewrite in Cython).

special

Though there are still a lot of functions that need improvements in precision, probably the only show-stoppers are hypergeometric functions, parabolic cylinder functions, and spheroidal wave functions. Three possible ways to handle this:

1. Get good double-precision implementations. This is doable for parabolic cylinder functions (in progress). I think it's possible for hypergeometric functions, though maybe not in time. For spheroidal wavefunctions this is not possible with current theory.
2. Port Boost's arbitrary precision library and use it under the hood to get double precision accuracy. This might be necessary as a stopgap measure for hypergeometric functions; the idea of using arbitrary precision has been suggested before by @nmayorov and in gh-5349. Likely necessary for spheroidal wave functions, this could be reused: https://github.com/radelman/scattering.
3. Add clear warnings to the documentation about the limits of the existing implementations.

stats

`stats.distributions` is in good shape.

`gaussian_kde` is in good shape but limited. It should not be expanded probably, this fits better in Statsmodels (which already has a lot more KDE functionality).

stats.mstats is a useful module for worked with data with missing values. One problem it has though is that in many cases the functions have diverged from their counterparts in scipy.stats. The mstats functions should be updated so that the two sets of functions are consistent.

weave

This module is deprecated and will be removed for Scipy 1.0. It has been packaged as a separate package "weave", which users that still rely on the functionality provided by scipy.weave should use.

Also note that this is the only module that was not ported to Python 3.

4.3.3 New modules under discussion

diff

Currently Scipy doesn't provide much support for numerical differentiation. A new scipy.diff module for that is discussed in https://github.com/scipy/scipy/issues/2035. There's also a fairly detailed GSoC proposal to build on, see here.

There is also approx_derivative in optimize, which is still private but could form a solid basis for this module.

transforms

This module was discussed previously, mainly to provide a home for discrete wavelet transform functionality. Other transforms could fit as well, for example there's a PR for a Hankel transform . *Note: this is on the back burner, because the plans to integrate PyWavelets DWT code has been put on hold.*

CHAPTER

FIVE

API REFERENCE

The exact API of all functions and classes, as given by the docstrings. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

5.1 Clustering package (`scipy.cluster`)

scipy.cluster.vq

Clustering algorithms are useful in information theory, target detection, communications, compression, and other areas. The *vq* module only supports vector quantization and the k-means algorithms.

scipy.cluster.hierarchy

The *hierarchy* module provides functions for hierarchical and agglomerative clustering. Its features include generating hierarchical clusters from distance matrices, calculating statistics on clusters, cutting linkages to generate flat clusters, and visualizing clusters with dendrograms.

5.2 K-means clustering and vector quantization (`scipy.cluster.vq`)

Provides routines for k-means clustering, generating code books from k-means models, and quantizing vectors by comparing them with centroids in a code book.

whiten(obs[, check_finite])	Normalize a group of observations on a per feature basis.
vq(obs, code_book[, check_finite])	Assign codes from a code book to observations.
kmeans(obs, k_or_guess[, iter, thresh, ...])	Performs k-means on a set of observation vectors forming k clusters.
kmeans2(data, k[, iter, thresh, minit, ...])	Classify a set of observations into k clusters using the k-means algorithm.

`scipy.cluster.vq.`**`whiten`**(*obs*, *check_finite=True*)

Normalize a group of observations on a per feature basis.

Before running k-means, it is beneficial to rescale each feature dimension of the observation set with whitening. Each feature is divided by its standard deviation across all observations to give it unit variance.

Parameters **obs** : ndarray

Each row of the array is an observation. The columns are the features seen during each observation.

```
>>> #         f0     f1     f2
>>> obs = [[  1.,    1.,    1.],  #o0
...        [  2.,    2.,    2.],  #o1
...        [  3.,    3.,    3.],  #o2
...        [  4.,    4.,    4.]]  #o3
```

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True

Returns **result** : ndarray
Contains the values in *obs* scaled by the standard deviation of each column.

Examples

```
>>> from scipy.cluster.vq import whiten
>>> features  = np.array([[1.9, 2.3, 1.7],
...                       [1.5, 2.5, 2.2],
...                       [0.8, 0.6, 1.7,]])
>>> whiten(features)
array([[ 4.17944278,  2.69811351,  7.21248917],
       [ 3.29956009,  2.93273208,  9.33380951],
       [ 1.75976538,  0.7038557 ,  7.21248917]])
```

`scipy.cluster.vq.`**`vq`**(*obs*, *code_book*, *check_finite=True*)
Assign codes from a code book to observations.

Assigns a code from a code book to each observation. Each observation vector in the 'M' by 'N' *obs* array is compared with the centroids in the code book and assigned the code of the closest centroid.

The features in *obs* should have unit variance, which can be achieved by passing them through the whiten function. The code book can be created with the k-means algorithm or a different encoding algorithm.

Parameters **obs** : ndarray
Each row of the 'M' x 'N' array is an observation. The columns are the "features" seen during each observation. The features must be whitened first using the whiten function or something equivalent.

code_book : ndarray
The code book is usually generated using the k-means algorithm. Each row of the array holds a different code, and the columns are the features of the code.

```
>>> #              f0     f1     f2    f3
>>> code_book = [
...             [  1.,    2.,    3.,    4.],  #c0
...             [  1.,    2.,    3.,    4.],  #c1
...             [  1.,    2.,    3.,    4.]]  #c2
```

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True

Returns **code** : ndarray
A length M array holding the code book index for each observation.

dist : ndarray
The distortion (distance) between the observation and its nearest code.

Examples

```
>>> from numpy import array
>>> from scipy.cluster.vq import vq
>>> code_book = array([[1.,1.,1.],
...                    [2.,2.,2.]])
>>> features  = array([[  1.9,2.3,1.7],
...                    [  1.5,2.5,2.2],
...                    [  0.8,0.6,1.7]])
>>> vq(features,code_book)
(array([1, 1, 0],'i'), array([ 0.43588989,  0.73484692,  0.83066239]))
```

`scipy.cluster.vq.`**`kmeans`**(*obs*, *k_or_guess*, *iter=20*, *thresh=1e-05*, *check_finite=True*)

Performs k-means on a set of observation vectors forming k clusters.

The k-means algorithm adjusts the centroids until sufficient progress cannot be made, i.e. the change in distortion since the last iteration is less than some threshold. This yields a code book mapping centroids to codes and vice versa.

Distortion is defined as the sum of the squared differences between the observations and the corresponding centroid.

Parameters

obs : ndarray
Each row of the M by N array is an observation vector. The columns are the features seen during each observation. The features must be whitened first with the *`whiten`* function.

k_or_guess : int or ndarray
The number of centroids to generate. A code is assigned to each centroid, which is also the row index of the centroid in the code_book matrix generated.
The initial k centroids are chosen by randomly selecting observations from the observation matrix. Alternatively, passing a k by N array specifies the initial k centroids.

iter : int, optional
The number of times to run k-means, returning the codebook with the lowest distortion. This argument is ignored if initial centroids are specified with an array for the `k_or_guess` parameter. This parameter does not represent the number of iterations of the k-means algorithm.

thresh : float, optional
Terminates the k-means algorithm if the change in distortion since the last k-means iteration is less than or equal to thresh.

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True

Returns

codebook : ndarray
A k by N array of k centroids. The i'th centroid codebook[i] is represented with the code i. The centroids and codes generated represent the lowest distortion seen, not necessarily the globally minimal distortion.

distortion : float
The distortion between the observations passed and the centroids generated.

See also:

`kmeans2` a different implementation of k-means clustering with more methods for generating initial centroids but without using a distortion change threshold as a stopping criterion.

`whiten` must be called prior to passing an observation matrix to kmeans.

Examples

```
>>> from numpy import array
>>> from scipy.cluster.vq import vq, kmeans, whiten
>>> features  = array([[ 1.9,2.3],
...                    [ 1.5,2.5],
...                    [ 0.8,0.6],
...                    [ 0.4,1.8],
...                    [ 0.1,0.1],
...                    [ 0.2,1.8],
...                    [ 2.0,0.5],
...                    [ 0.3,1.5],
...                    [ 1.0,1.0]])
>>> whitened = whiten(features)
>>> book = np.array((whitened[0],whitened[2]))
>>> kmeans(whitened,book)
(array([[ 2.3110306 ,  2.86287398],    # random
       [ 0.93218041,  1.24398691]]), 0.85684700941625547)
```

```
>>> from numpy import random
>>> random.seed((1000,2000))
>>> codes = 3
>>> kmeans(whitened,codes)
(array([[ 2.3110306 ,  2.86287398],    # random
       [ 1.32544402,  0.65607529],
       [ 0.40782893,  2.02786907]]), 0.5196582527686241)
```

`scipy.cluster.vq.`**`kmeans2`**(*data*, *k*, *iter=10*, *thresh=1e-05*, *minit='random'*, *missing='warn'*, *check_finite=True*)

Classify a set of observations into k clusters using the k-means algorithm.

The algorithm attempts to minimize the Euclidian distance between observations and centroids. Several initialization methods are included.

Parameters **data** : ndarray

A 'M' by 'N' array of 'M' observations in 'N' dimensions or a length 'M' array of 'M' one-dimensional observations.

k : int or ndarray

The number of clusters to form as well as the number of centroids to generate. If *minit* initialization string is 'matrix', or if a ndarray is given instead, it is interpreted as initial cluster to use instead.

iter : int, optional

Number of iterations of the k-means algrithm to run. Note that this differs in meaning from the iters parameter to the kmeans function.

thresh : float, optional

(not used yet)

minit : str, optional

Method for initialization. Available methods are 'random', 'points', and 'matrix':
'random': generate k centroids from a Gaussian with mean and variance estimated from the data.
'points': choose k observations (rows) at random from data for the initial centroids.
'matrix': interpret the k parameter as a k by M (or length k array for one-dimensional data) array of initial centroids.

missing : str, optional

Method to deal with empty clusters. Available methods are 'warn' and 'raise':
'warn': give a warning and continue.
'raise': raise an ClusterError and terminate the algorithm.

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True

Returns
centroid : ndarray
A 'k' by 'N' array of centroids found at the last iteration of k-means.
label : ndarray
label[i] is the code or index of the centroid the i'th observation is closest to.

5.2.1 Background information

The k-means algorithm takes as input the number of clusters to generate, k, and a set of observation vectors to cluster. It returns a set of centroids, one for each of the k clusters. An observation vector is classified with the cluster number or centroid index of the centroid closest to it.

A vector v belongs to cluster i if it is closer to centroid i than any other centroids. If v belongs to i, we say centroid i is the dominating centroid of v. The k-means algorithm tries to minimize distortion, which is defined as the sum of the squared distances between each observation vector and its dominating centroid. Each step of the k-means algorithm refines the choices of centroids to reduce distortion. The change in distortion is used as a stopping criterion: when the change is lower than a threshold, the k-means algorithm is not making sufficient progress and terminates. One can also define a maximum number of iterations.

Since vector quantization is a natural application for k-means, information theory terminology is often used. The centroid index or cluster index is also referred to as a "code" and the table mapping codes to centroids and vice versa is often referred as a "code book". The result of k-means, a set of centroids, can be used to quantize vectors. Quantization aims to find an encoding of vectors that reduces the expected distortion.

All routines expect obs to be a M by N array where the rows are the observation vectors. The codebook is a k by N array where the i'th row is the centroid of code word i. The observation vectors and centroids have the same feature dimension.

As an example, suppose we wish to compress a 24-bit color image (each pixel is represented by one byte for red, one for blue, and one for green) before sending it over the web. By using a smaller 8-bit encoding, we can reduce the amount of data by two thirds. Ideally, the colors for each of the 256 possible 8-bit encoding values should be chosen to minimize distortion of the color. Running k-means with k=256 generates a code book of 256 codes, which fills up all possible 8-bit sequences. Instead of sending a 3-byte value for each pixel, the 8-bit centroid index (or code word) of the dominating centroid is transmitted. The code book is also sent over the wire so each 8-bit code can be translated back to a 24-bit pixel value representation. If the image of interest was of an ocean, we would expect many 24-bit blues to be represented by 8-bit codes. If it was an image of a human face, more flesh tone colors would be represented in the code book.

5.3 Hierarchical clustering (`scipy.cluster.hierarchy`)

These functions cut hierarchical clusterings into flat clusterings or find the roots of the forest formed by a cut by providing the flat cluster ids of each observation.

fcluster(Z, t[, criterion, depth, R, monocrit])	Forms flat clusters from the hierarchical clustering defined by the given linkage matrix.
fclusterdata(X, t[, criterion, metric, ...])	Cluster observation data using a given metric.
leaders(Z, T)	Returns the root nodes in a hierarchical clustering.

scipy.cluster.hierarchy.**fcluster**(*Z, t, criterion='inconsistent', depth=2, R=None, monocrit=None*)

Forms flat clusters from the hierarchical clustering defined by the given linkage matrix.

Parameters **Z** : ndarray

The hierarchical clustering encoded with the matrix returned by the `linkage` function.

t : float

The threshold to apply when forming flat clusters.

criterion : str, optional

The criterion to use in forming flat clusters. This can be any of the following values:

inconsistent

[If a cluster node and all its] descendants have an inconsistent value less than or equal to *t* then all its leaf descendants belong to the same flat cluster. When no non-singleton cluster meets this criterion, every node is assigned to its own cluster. (Default)

distance [Forms flat clusters so that the original] observations in each flat cluster have no greater a cophenetic distance than *t*.

maxclust [Finds a minimum threshold `r` so that] the cophenetic distance between any two original observations in the same flat cluster is no more than `r` and no more than *t* flat clusters are formed.

monocrit [Forms a flat cluster from a cluster node c] with index i when `monocrit[j] <= t`.

For example, to threshold on the maximum mean distance as computed in the inconsistency matrix R with a threshold of 0.8 do:

```
MR = maxRstat(Z, R, 3)
cluster(Z, t=0.8, criterion='monocrit',␣
↪monocrit=MR)
```

maxclust_monocrit

[Forms a flat cluster from a] non-singleton cluster node `c` when `monocrit[i] <= r` for all cluster indices `i` below and including `c`. `r` is minimized such that no more than `t` flat clusters are formed. monocrit must be monotonic. For example, to minimize the threshold t on maximum inconsistency values so that no more than 3 flat clusters are formed, do:

```
MI = maxinconsts(Z, R)
cluster(Z, t=3, criterion='maxclust_monocrit',␣
↪monocrit=MI)
```

depth : int, optional

The maximum depth to perform the inconsistency calculation. It has no meaning for the other criteria. Default is 2.

R : ndarray, optional

The inconsistency matrix to use for the 'inconsistent' criterion. This matrix is computed if not provided.

monocrit : ndarray, optional

An array of length n-1. *monocrit[i]* is the statistics upon which non-singleton i is thresholded. The monocrit vector must be monotonic, i.e. given a node c with index i, for all node indices j corresponding to nodes below c, `monocrit[i] >= monocrit[j]`.

Returns **fcluster** : ndarray

An array of length `n`. `T[i]` is the flat cluster number to which original observation `i` belongs.

`scipy.cluster.hierarchy.`**`fclusterdata`**(*X*, *t*, *criterion='inconsistent'*, *metric='euclidean'*, *depth=2*, *method='single'*, *R=None*)

Cluster observation data using a given metric.

Clusters the original observations in the n-by-m data matrix X (n observations in m dimensions), using the euclidean distance metric to calculate distances between original observations, performs hierarchical clustering using the single linkage algorithm, and forms flat clusters using the inconsistency method with *t* as the cut-off threshold.

A one-dimensional array `T` of length `n` is returned. `T[i]` is the index of the flat cluster to which the original observation `i` belongs.

Parameters

X : (N, M) ndarray
: N by M data matrix with N observations in M dimensions.

t : float
: The threshold to apply when forming flat clusters.

criterion : str, optional
: Specifies the criterion for forming flat clusters. Valid values are 'inconsistent' (default), 'distance', or 'maxclust' cluster formation algorithms. See *`fcluster`* for descriptions.

metric : str, optional
: The distance metric for calculating pairwise distances. See `distance.pdist` for descriptions and linkage to verify compatibility with the linkage method.

depth : int, optional
: The maximum depth for the inconsistency calculation. See *`inconsistent`* for more information.

method : str, optional
: The linkage method to use (single, complete, average, weighted, median centroid, ward). See *`linkage`* for more information. Default is "single".

R : ndarray, optional
: The inconsistency matrix. It will be computed if necessary if it is not passed.

Returns

fclusterdata : ndarray
: A vector of length n. T[i] is the flat cluster number to which original observation i belongs.

See also:

`scipy.spatial.distance.pdist`
: pairwise distance metrics

Notes

This function is similar to the MATLAB function `clusterdata`.

`scipy.cluster.hierarchy.`**`leaders`**(*Z*, *T*)

Returns the root nodes in a hierarchical clustering.

Returns the root nodes in a hierarchical clustering corresponding to a cut defined by a flat cluster assignment vector `T`. See the `fcluster` function for more information on the format of `T`.

For each flat cluster j of the k flat clusters represented in the n-sized flat cluster assignment vector `T`, this function finds the lowest cluster node i in the linkage tree Z such that:

- leaf descendents belong only to flat cluster j (i.e. `T[p]==j` for all p in $S(i)$ where $S(i)$ is the set of leaf ids of leaf nodes descendent with cluster node i)
- there does not exist a leaf that is not descendent with i that also belongs to cluster j (i.e. `T[q]!=j` for all q not in $S(i)$). If this condition is violated, `T` is not a valid cluster assignment vector, and an exception will be thrown.

Parameters
Z : ndarray
The hierarchical clustering encoded as a matrix. See *linkage* for more information.
T : ndarray
The flat cluster assignment vector.

Returns
L : ndarray
The leader linkage node id's stored as a k-element 1-D array where `k` is the number of flat clusters found in `T`.
`L[j]=i` is the linkage cluster node id that is the leader of flat cluster with id M[j]. If `i < n`, `i` corresponds to an original observation, otherwise it corresponds to a non-singleton cluster.
For example: if `L[3]=2` and `M[3]=8`, the flat cluster with id 8's leader is linkage node 2.
M : ndarray
The leader linkage node id's stored as a k-element 1-D array where `k` is the number of flat clusters found in `T`. This allows the set of flat cluster ids to be any arbitrary set of `k` integers.

These are routines for agglomerative clustering.

linkage(y[, method, metric])	Performs hierarchical/agglomerative clustering.
single(y)	Performs single/min/nearest linkage on the condensed distance matrix `y`
complete(y)	Performs complete/max/farthest point linkage on a condensed distance matrix
average(y)	Performs average/UPGMA linkage on a condensed distance matrix
weighted(y)	Performs weighted/WPGMA linkage on the condensed distance matrix.
centroid(y)	Performs centroid/UPGMC linkage.
median(y)	Performs median/WPGMC linkage.
ward(y)	Performs Ward's linkage on a condensed distance matrix.

`scipy.cluster.hierarchy.`**`linkage`**(*y*, *method='single'*, *metric='euclidean'*)

Performs hierarchical/agglomerative clustering.

The input y may be either a 1d compressed distance matrix or a 2d array of observation vectors.

If y is a 1d compressed distance matrix, then y must be a $\binom{n}{2}$ sized vector where n is the number of original observations paired in the distance matrix. The behavior of this function is very similar to the MATLAB linkage function.

A $(n-1)$ by 4 matrix `Z` is returned. At the i-th iteration, clusters with indices `Z[i, 0]` and `Z[i, 1]` are combined to form cluster $n+i$. A cluster with an index less than n corresponds to one of the n original observations. The distance between clusters `Z[i, 0]` and `Z[i, 1]` is given by `Z[i, 2]`. The fourth value `Z[i, 3]` represents the number of original observations in the newly formed cluster.

The following linkage methods are used to compute the distance $d(s,t)$ between two clusters s and t. The algorithm begins with a forest of clusters that have yet to be used in the hierarchy being formed. When two clusters s and t from this forest are combined into a single cluster u, s and t are removed from the forest, and u is added to the forest. When only one cluster remains in the forest, the algorithm stops, and this cluster becomes the root.

A distance matrix is maintained at each iteration. The `d[i, j]` entry corresponds to the distance between cluster i and j in the original forest.

At each iteration, the algorithm must update the distance matrix to reflect the distance of the newly formed cluster u with the remaining clusters in the forest.

Suppose there are $|u|$ original observations $u[0], \ldots, u[|u|-1]$ in cluster u and $|v|$ original objects $v[0], \ldots, v[|v|-1]$ in cluster v. Recall s and t are combined to form cluster u. Let v be any remaining cluster in the forest that is not u.

The following are methods for calculating the distance between the newly formed cluster u and each v.

- method='single' assigns

  $$d(u,v) = \min(dist(u[i],v[j]))$$

  for all points i in cluster u and j in cluster v. This is also known as the Nearest Point Algorithm.

- method='complete' assigns

  $$d(u,v) = \max(dist(u[i],v[j]))$$

  for all points i in cluster u and j in cluster v. This is also known by the Farthest Point Algorithm or Voor Hees Algorithm.

- method='average' assigns

  $$d(u,v) = \sum_{ij} \frac{d(u[i],v[j])}{(|u| * |v|)}$$

  for all points i and j where $|u|$ and $|v|$ are the cardinalities of clusters u and v, respectively. This is also called the UPGMA algorithm.

- method='weighted' assigns

  $$d(u,v) = (dist(s,v) + dist(t,v))/2$$

  where cluster u was formed with cluster s and t and v is a remaining cluster in the forest. (also called WPGMA)

- method='centroid' assigns

  $$dist(s,t) = ||c_s - c_t||_2$$

  where c_s and c_t are the centroids of clusters s and t, respectively. When two clusters s and t are combined into a new cluster u, the new centroid is computed over all the original objects in clusters s and t. The distance then becomes the Euclidean distance between the centroid of u and the centroid of a remaining cluster v in the forest. This is also known as the UPGMC algorithm.

- method='median' assigns $d(s,t)$ like the `centroid` method. When two clusters s and t are combined into a new cluster u, the average of centroids s and t give the new centroid u. This is also known as the WPGMC algorithm.

- method='ward' uses the Ward variance minimization algorithm. The new entry $d(u,v)$ is computed as follows,

  $$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

  where u is the newly joined cluster consisting of clusters s and t, v is an unused cluster in the forest, $T = |v| + |s| + |t|$, and $| * |$ is the cardinality of its argument. This is also known as the incremental algorithm.

Warning: When the minimum distance pair in the forest is chosen, there may be two or more pairs with the same minimum distance. This implementation may chose a different minimum than the MATLAB version.

Parameters **y** : ndarray

A condensed distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as an m by n array. All elements of the condensed distance matrix must be finite, i.e. no NaNs or infs.

method : str, optional

The linkage algorithm to use. See the `Linkage Methods` section below for full descriptions.

metric : str or function, optional

The distance metric to use in the case that y is a collection of observation vectors; ignored otherwise. See the `pdist` function for a list of valid distance metrics. A custom distance function can also be used.

Returns **Z** : ndarray

The hierarchical clustering encoded as a linkage matrix.

See also:

`scipy.spatial.distance.pdist`
pairwise distance metrics

Notes

1.For method 'single' an optimized algorithm based on minimum spanning tree is implemented. It has time complexity $O(n^2)$. For methods 'complete', 'average', 'weighted' and 'ward' an algorithm called nearest-neighbors chain is implemented. It also has time complexity $O(n^2)$. For other methods a naive algorithm is implemented with $O(n^3)$ time complexity. All algorithms use $O(n^2)$ memory. Refer to *[R41]* for details about the algorithms.

2.Methods 'centroid', 'median' and 'ward' are correctly defined only if Euclidean pairwise metric is used. If *y* is passed as precomputed pairwise distances, then it is a user responsibility to assure that these distances are in fact Euclidean, otherwise the produced result will be incorrect.

References

[R41]

`scipy.cluster.hierarchy.`**`single`**`(y)`

Performs single/min/nearest linkage on the condensed distance matrix `y`

Parameters **y** : ndarray

The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns **Z** : ndarray

The linkage matrix.

See also:

`linkage` for advanced creation of hierarchical clusterings.

`scipy.spatial.distance.pdist`
pairwise distance metrics

`scipy.cluster.hierarchy.`**`complete`**`(y)`

Performs complete/max/farthest point linkage on a condensed distance matrix

Parameters **y** : ndarray
The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns **Z** : ndarray
A linkage matrix containing the hierarchical clustering. See the *linkage* function documentation for more information on its structure.

See also:

linkage for advanced creation of hierarchical clusterings.

scipy.spatial.distance.pdist
pairwise distance metrics

`scipy.cluster.hierarchy.`**`average`**`(`*y*`)`

Performs average/UPGMA linkage on a condensed distance matrix

Parameters **y** : ndarray
The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns **Z** : ndarray
A linkage matrix containing the hierarchical clustering. See *linkage* for more information on its structure.

See also:

linkage for advanced creation of hierarchical clusterings.

scipy.spatial.distance.pdist
pairwise distance metrics

`scipy.cluster.hierarchy.`**`weighted`**`(`*y*`)`

Performs weighted/WPGMA linkage on the condensed distance matrix.

See *linkage* for more information on the return structure and algorithm.

Parameters **y** : ndarray
The upper triangular of the distance matrix. The result of `pdist` is returned in this form.

Returns **Z** : ndarray
A linkage matrix containing the hierarchical clustering. See *linkage* for more information on its structure.

See also:

linkage for advanced creation of hierarchical clusterings.

scipy.spatial.distance.pdist
pairwise distance metrics

`scipy.cluster.hierarchy.`**`centroid`**`(`*y*`)`

Performs centroid/UPGMC linkage.

See *linkage* for more information on the input matrix, return structure, and algorithm.

The following are common calling conventions:

1. `Z = centroid(y)`

   Performs centroid/UPGMC linkage on the condensed distance matrix `y`.

2. `Z = centroid(X)`

Performs centroid/UPGMC linkage on the observation matrix `X` using Euclidean distance as the distance metric.

Parameters **y** : ndarray
A condensed distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns **Z** : ndarray
A linkage matrix containing the hierarchical clustering. See the *linkage* function documentation for more information on its structure.

See also:

linkage for advanced creation of hierarchical clusterings.

`scipy.cluster.hierarchy.`**`median`**`(y)`

Performs median/WPGMC linkage.

See *linkage* for more information on the return structure and algorithm.

The following are common calling conventions:

1. `Z = median(y)`
Performs median/WPGMC linkage on the condensed distance matrix `y`. See `linkage` for more information on the return structure and algorithm.
2. `Z = median(X)`
Performs median/WPGMC linkage on the observation matrix `X` using Euclidean distance as the distance metric. See *linkage* for more information on the return structure and algorithm.

Parameters **y** : ndarray
A condensed distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns **Z** : ndarray
The hierarchical clustering encoded as a linkage matrix.

See also:

linkage for advanced creation of hierarchical clusterings.

scipy.spatial.distance.pdist
pairwise distance metrics

`scipy.cluster.hierarchy.`**`ward`**`(y)`

Performs Ward's linkage on a condensed distance matrix.

See *linkage* for more information on the return structure and algorithm.

The following are common calling conventions:

1. `Z = ward(y)` Performs Ward's linkage on the condensed distance matrix `y`.
2. `Z = ward(X)` Performs Ward's linkage on the observation matrix `X` using Euclidean distance as the distance metric.

Parameters **y** : ndarray

A condensed distance matrix. A condensed distance matrix is a flat array containing the upper triangular of the distance matrix. This is the form that `pdist` returns. Alternatively, a collection of m observation vectors in n dimensions may be passed as a m by n array.

Returns **Z** : ndarray

The hierarchical clustering encoded as a linkage matrix. See *linkage* for more information on the return structure and algorithm.

See also:

linkage for advanced creation of hierarchical clusterings.

scipy.spatial.distance.pdist
pairwise distance metrics

These routines compute statistics on hierarchies.

cophenet(Z[, Y])	Calculates the cophenetic distances between each observation in the hierarchical clustering defined by the linkage Z.
from_mlab_linkage(Z)	Converts a linkage matrix generated by MATLAB(TM) to a new linkage matrix compatible with this module.
inconsistent(Z[, d])	Calculates inconsistency statistics on a linkage matrix.
maxinconsts(Z, R)	Returns the maximum inconsistency coefficient for each non-singleton cluster and its descendents.
maxdists(Z)	Returns the maximum distance between any non-singleton cluster.
maxRstat(Z, R, i)	Returns the maximum statistic for each non-singleton cluster and its descendents.
to_mlab_linkage(Z)	Converts a linkage matrix to a MATLAB(TM) compatible one.

scipy.cluster.hierarchy.**cophenet**(*Z*, *Y=None*)

Calculates the cophenetic distances between each observation in the hierarchical clustering defined by the linkage Z.

Suppose p and q are original observations in disjoint clusters s and t, respectively and s and t are joined by a direct parent cluster u. The cophenetic distance between observations i and j is simply the distance between clusters s and t.

Parameters **Z** : ndarray

The hierarchical clustering encoded as an array (see *linkage* function).

Y : ndarray (optional)

Calculates the cophenetic correlation coefficient c of a hierarchical clustering defined by the linkage matrix *Z* of a set of n observations in m dimensions. *Y* is the condensed distance matrix from which *Z* was generated.

Returns **c** : ndarray

The cophentic correlation distance (if Y is passed).

d : ndarray

The cophenetic distance matrix in condensed form. The ij th entry is the cophenetic distance between original observations i and j.

scipy.cluster.hierarchy.**from_mlab_linkage**(*Z*)

Converts a linkage matrix generated by MATLAB(TM) to a new linkage matrix compatible with this module.

The conversion does two things:

- the indices are converted from `1..N` to `0..(N-1)` form, and
- a fourth column `Z[:,3]` is added where `Z[i,3]` represents the number of original observations (leaves) in the non-singleton cluster `i`.

This function is useful when loading in linkages from legacy data files generated by MATLAB.

Parameters **Z** : ndarray
A linkage matrix generated by MATLAB(TM).

Returns **ZS** : ndarray
A linkage matrix compatible with `scipy.cluster.hierarchy`.

`scipy.cluster.hierarchy.`**`inconsistent`**(*Z*, *d=2*)

Calculates inconsistency statistics on a linkage matrix.

Parameters **Z** : ndarray
The $(n-1)$ by 4 matrix encoding the linkage (hierarchical clustering). See *linkage* documentation for more information on its form.

d : int, optional
The number of links up to *d* levels below each non-singleton cluster.

Returns **R** : ndarray
A $(n-1)$ by 5 matrix where the `i`'th row contains the link statistics for the non-singleton cluster `i`. The link statistics are computed over the link heights for links d levels below the cluster `i`. `R[i,0]` and `R[i,1]` are the mean and standard deviation of the link heights, respectively; `R[i,2]` is the number of links included in the calculation; and `R[i,3]` is the inconsistency coefficient,

$$\frac{\mathtt{Z[i,2]} - \mathtt{R[i,0]}}{R[i,1]}$$

Notes

This function behaves similarly to the MATLAB(TM) `inconsistent` function.

`scipy.cluster.hierarchy.`**`maxinconsts`**(*Z*, *R*)

Returns the maximum inconsistency coefficient for each non-singleton cluster and its descendents.

Parameters **Z** : ndarray
The hierarchical clustering encoded as a matrix. See *linkage* for more information.

R : ndarray
The inconsistency matrix.

Returns **MI** : ndarray
A monotonic `(n-1)`-sized numpy array of doubles.

`scipy.cluster.hierarchy.`**`maxdists`**(*Z*)

Returns the maximum distance between any non-singleton cluster.

Parameters **Z** : ndarray
The hierarchical clustering encoded as a matrix. See `linkage` for more information.

Returns **maxdists** : ndarray
A `(n-1)` sized numpy array of doubles; `MD[i]` represents the maximum distance between any cluster (including singletons) below and including the node with index i. More specifically, `MD[i] = Z[Q(i)-n, 2].max()` where `Q(i)` is the set of all node indices below and including node i.

`scipy.cluster.hierarchy.`**`maxRstat`**(*Z*, *R*, *i*)

Returns the maximum statistic for each non-singleton cluster and its descendents.

Parameters **Z** : array_like
The hierarchical clustering encoded as a matrix. See *linkage* for more information.

R : array_like
The inconsistency matrix.

i : int
The column of *R* to use as the statistic.

Returns **MR** : ndarray
Calculates the maximum statistic for the i'th column of the inconsistency matrix *R* for each non-singleton cluster node. `MR[j]` is the maximum over `R[Q(j)-n, i]` where `Q(j)` the set of all node ids corresponding to nodes below and including `j`.

`scipy.cluster.hierarchy.`**`to_mlab_linkage`**`(`*Z*`)`

Converts a linkage matrix to a MATLAB(TM) compatible one.

Converts a linkage matrix `Z` generated by the linkage function of this module to a MATLAB(TM) compatible one. The return linkage matrix has the last column removed and the cluster indices are converted to `1..N` indexing.

Parameters **Z** : ndarray
A linkage matrix generated by `scipy.cluster.hierarchy`.

Returns **to_mlab_linkage** : ndarray
A linkage matrix compatible with MATLAB(TM)'s hierarchical clustering functions. The return linkage matrix has the last column removed and the cluster indices are converted to `1..N` indexing.

Routines for visualizing flat clusters.

dendrogram(Z[, p, truncate_mode, ...])	Plots the hierarchical clustering as a dendrogram.

`scipy.cluster.hierarchy.`**`dendrogram`**`(`*Z*, *p=30*, *truncate_mode=None*, *color_threshold=None*, *get_leaves=True*, *orientation='top'*, *labels=None*, *count_sort=False*, *distance_sort=False*, *show_leaf_counts=True*, *no_plot=False*, *no_labels=False*, *leaf_font_size=None*, *leaf_rotation=None*, *leaf_label_func=None*, *show_contracted=False*, *link_color_func=None*, *ax=None*, *above_threshold_color='b'*`)`

Plots the hierarchical clustering as a dendrogram.

The dendrogram illustrates how each cluster is composed by drawing a U-shaped link between a non-singleton cluster and its children. The top of the U-link indicates a cluster merge. The two legs of the U-link indicate which clusters were merged. The length of the two legs of the U-link represents the distance between the child clusters. It is also the cophenetic distance between original observations in the two children clusters.

Parameters **Z** : ndarray
The linkage matrix encoding the hierarchical clustering to render as a dendrogram. See the `linkage` function for more information on the format of `Z`.

p : int, optional
The `p` parameter for `truncate_mode`.

truncate_mode : str, optional
The dendrogram can be hard to read when the original observation matrix from which the linkage is derived is large. Truncation is used to condense the dendrogram. There are several modes:

`None` No truncation is performed (default). Note: `'none'` is an alias for `None` that's kept for backward compatibility.

`'lastp'` The last `p` non-singleton clusters formed in the linkage are the only non-leaf nodes in the linkage; they correspond to rows `Z[n-p-2:end]` in `Z`. All other non-singleton clusters are contracted into leaf nodes.

'level' No more than p levels of the dendrogram tree are displayed. A "level" includes all nodes with p merges from the last merge.
Note: 'mtica' is an alias for 'level' that's kept for backward compatibility.

color_threshold : double, optional
For brevity, let t be the color_threshold. Colors all the descendent links below a cluster node k the same color if k is the first node below the cut threshold t. All links connecting nodes with distances greater than or equal to the threshold are colored blue. If t is less than or equal to zero, all nodes are colored blue. If color_threshold is None or 'default', corresponding with MATLAB(TM) behavior, the threshold is set to 0.7*max(Z[:,2]).

get_leaves : bool, optional
Includes a list R['leaves']=H in the result dictionary. For each i, H[i] == j, cluster node j appears in position i in the left-to-right traversal of the leaves, where $j < 2n - 1$ and $i < n$.

orientation : str, optional
The direction to plot the dendrogram, which can be any of the following strings:

'top' Plots the root at the top, and plot descendent links going downwards. (default).

'bottom' Plots the root at the bottom, and plot descendent links going upwards.

'left' Plots the root at the left, and plot descendent links going right.

'right' Plots the root at the right, and plot descendent links going left.

labels : ndarray, optional
By default labels is None so the index of the original observation is used to label the leaf nodes. Otherwise, this is an n -sized list (or tuple). The labels[i] value is the text to put under the i th leaf node only if it corresponds to an original observation and not a non-singleton cluster.

count_sort : str or bool, optional
For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:

False Nothing is done.

'ascending' or True
The child with the minimum number of original objects in its cluster is plotted first.

'descendent'
The child with the maximum number of original objects in its cluster is plotted first.

Note distance_sort and count_sort cannot both be True.

distance_sort : str or bool, optional
For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:

False Nothing is done.

'ascending' or True
The child with the minimum distance between its direct descendents is plotted first.

'descending'
The child with the maximum distance between its direct descendents is plotted first.

Note distance_sort and count_sort cannot both be True.

show_leaf_counts : bool, optional
When True, leaf nodes representing $k > 1$ original observation are labeled with the number of observations they contain in parentheses.

no_plot : bool, optional

When True, the final rendering is not performed. This is useful if only the data structures computed for the rendering are needed or if matplotlib is not available.

no_labels : bool, optional
When True, no labels appear next to the leaf nodes in the rendering of the dendrogram.

leaf_rotation : double, optional
Specifies the angle (in degrees) to rotate the leaf labels. When unspecified, the rotation is based on the number of nodes in the dendrogram (default is 0).

leaf_font_size : int, optional
Specifies the font size (in points) of the leaf labels. When unspecified, the size based on the number of nodes in the dendrogram.

leaf_label_func : lambda or function, optional
When leaf_label_func is a callable function, for each leaf with cluster index $k < 2n-1$. The function is expected to return a string with the label for the leaf.
Indices $k < n$ correspond to original observations while indices $k \geq n$ correspond to non-singleton clusters.
For example, to label singletons with their node id and non-singletons with their id, count, and inconsistency coefficient, simply do:

```
# First define the leaf label function.
def llf(id):
    if id < n:
        return str(id)
    else:
        return '[%d %d %1.2f]' % (id, count, R[n-id,3])
# The text for the leaf nodes is going to be big so force
# a rotation of 90 degrees.
dendrogram(Z, leaf_label_func=llf, leaf_rotation=90)
```

show_contracted : bool, optional
When True the heights of non-singleton nodes contracted into a leaf node are plotted as crosses along the link connecting that leaf node. This really is only useful when truncation is used (see `truncate_mode` parameter).

link_color_func : callable, optional
If given, *link_color_function* is called with each non-singleton id corresponding to each U-shaped link it will paint. The function is expected to return the color to paint the link, encoded as a matplotlib color string code. For example:

```
dendrogram(Z, link_color_func=lambda k: colors[k])
```

colors the direct links below each untruncated non-singleton node `k` using `colors[k]`.

ax : matplotlib Axes instance, optional
If None and *no_plot* is not True, the dendrogram will be plotted on the current axes. Otherwise if *no_plot* is not True the dendrogram will be plotted on the given `Axes` instance. This can be useful if the dendrogram is part of a more complex figure.

above_threshold_color : str, optional
This matplotlib color string sets the color of the links above the color_threshold. The default is 'b'.

Returns **R** : dict
A dictionary of data structures computed to render the dendrogram. Its has the following keys:

'color_list'
A list of color names. The k'th element represents the color of the k'th link.

'icoord'* and *'dcoord'
Each of them is a list of lists. Let `icoord = [I1, I2, ...`

, `Ip]` where `Ik = [xk1, xk2, xk3, xk4]` and `dcoord = [D1, D2, ..., Dp]` where `Dk = [yk1, yk2, yk3, yk4]`, then the k'th link painted is `(xk1, yk1) - (xk2, yk2) - (xk3, yk3) - (xk4, yk4)`.

'ivl' A list of labels corresponding to the leaf nodes.

'leaves' For each i, `H[i] == j`, cluster node `j` appears in position `i` in the left-to-right traversal of the leaves, where $j < 2n - 1$ and $i < n$. If `j` is less than `n`, the `i`-th leaf node corresponds to an original observation. Otherwise, it corresponds to a non-singleton cluster.

See also:

linkage, *set_link_color_palette*

Notes

It is expected that the distances in `Z[:,2]` be monotonic, otherwise crossings appear in the dendrogram.

Examples

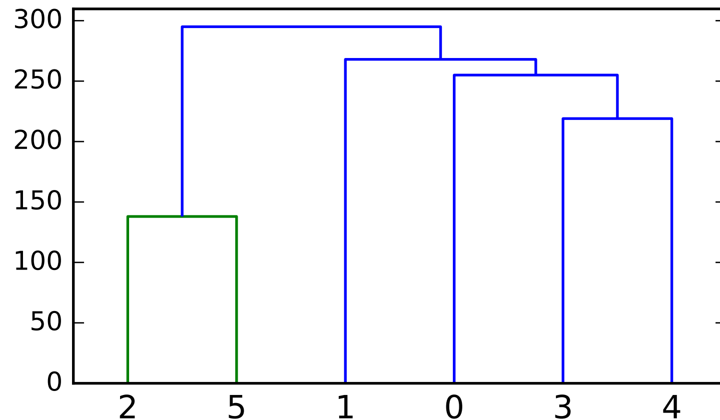
```
>>> from scipy.cluster import hierarchy
>>> import matplotlib.pyplot as plt
```

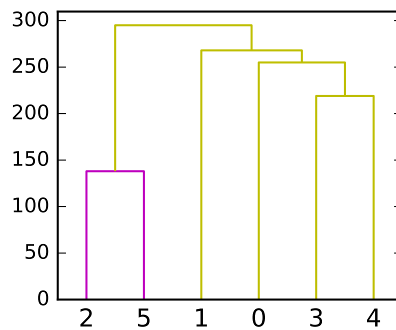
A very basic example:

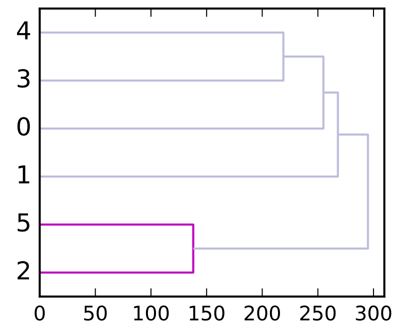
```
>>> ytdist = np.array([662., 877., 255., 412., 996., 295., 468., 268.,
...                    400., 754., 564., 138., 219., 869., 669.])
>>> Z = hierarchy.linkage(ytdist, 'single')
>>> plt.figure()
>>> dn = hierarchy.dendrogram(Z)
```

Now plot in given axes, improve the color scheme and use both vertical and horizontal orientations:

```
>>> hierarchy.set_link_color_palette(['m', 'c', 'y', 'k'])
>>> fig, axes = plt.subplots(1, 2, figsize=(8, 3))
>>> dn1 = hierarchy.dendrogram(Z, ax=axes[0], above_threshold_color='y',
...                            orientation='top')
>>> dn2 = hierarchy.dendrogram(Z, ax=axes[1], above_threshold_color='#bcbddc',
...                            orientation='right')
>>> hierarchy.set_link_color_palette(None)  # reset to default after use
>>> plt.show()
```







These are data structures and routines for representing hierarchies as tree objects.

`ClusterNode`(id[, left, right, dist, count])	A tree node class for representing a cluster.
`leaves_list`(Z)	Returns a list of leaf node ids
`to_tree`(Z[, rd])	Converts a linkage matrix into an easy-to-use tree object.
`cut_tree`(Z[, n_clusters, height])	Given a linkage matrix Z, return the cut tree.

class `scipy.cluster.hierarchy.`**`ClusterNode`** (*id*, *left=None*, *right=None*, *dist=0*, *count=1*)

A tree node class for representing a cluster.

Leaf nodes correspond to original observations, while non-leaf nodes correspond to non-singleton clusters.

The `to_tree` function converts a matrix returned by the linkage function into an easy-to-use tree representation.

All parameter names are also attributes.

Parameters **id** : int
: The node id.

left : ClusterNode instance, optional
: The left child tree node.

right : ClusterNode instance, optional

The right child tree node.

dist : float, optional
Distance for this cluster in the linkage matrix.

count : int, optional
The number of samples in this cluster.

See also:

to_tree for converting a linkage matrix Z into a tree object.

Methods

get_count()	The number of leaf nodes (original observations) belonging to the cluster node nd.
get_id()	The identifier of the target node.
get_left()	Return a reference to the left child tree object.
get_right()	Returns a reference to the right child tree object.
is_leaf()	Returns True if the target node is a leaf.
pre_order([func])	Performs pre-order traversal without recursive function calls.

ClusterNode.**get_count**()
The number of leaf nodes (original observations) belonging to the cluster node nd. If the target node is a leaf, 1 is returned.

Returns **get_count** : int
The number of leaf nodes below the target node.

ClusterNode.**get_id**()
The identifier of the target node.

For `0 <= i < n`, *i* corresponds to original observation i. For `n <= i < 2n-1`, *i* corresponds to non-singleton cluster formed at iteration `i-n`.

Returns **id** : int
The identifier of the target node.

ClusterNode.**get_left**()
Return a reference to the left child tree object.

Returns **left** : ClusterNode
The left child of the target node. If the node is a leaf, None is returned.

ClusterNode.**get_right**()
Returns a reference to the right child tree object.

Returns **right** : ClusterNode
The left child of the target node. If the node is a leaf, None is returned.

ClusterNode.**is_leaf**()
Returns True if the target node is a leaf.

Returns **leafness** : bool
True if the target node is a leaf node.

ClusterNode.**pre_order**(*func=<function <lambda>>*)
Performs pre-order traversal without recursive function calls.

When a leaf node is first encountered, `func` is called with the leaf node as its argument, and its result is appended to the list.

For example, the statement:

```
ids = root.pre_order(lambda x: x.id)
```

returns a list of the node ids corresponding to the leaf nodes of the tree as they appear from left to right.

Parameters **func** : function
Applied to each leaf ClusterNode object in the pre-order traversal. Given the `i`-th leaf node in the pre-order traversal `n[i]`, the result of `func(n[i])` is stored in `L[i]`. If not provided, the index of the original observation to which the node corresponds is used.

Returns **L** : list
The pre-order traversal.

`scipy.cluster.hierarchy.`**`leaves_list`**`(`*Z*`)`

Returns a list of leaf node ids

The return corresponds to the observation vector index as it appears in the tree from left to right. Z is a linkage matrix.

Parameters **Z** : ndarray
The hierarchical clustering encoded as a matrix. *Z* is a linkage matrix. See *`linkage`* for more information.

Returns **leaves_list** : ndarray
The list of leaf node ids.

`scipy.cluster.hierarchy.`**`to_tree`**`(`*Z*, *rd=False*`)`

Converts a linkage matrix into an easy-to-use tree object.

The reference to the root *`ClusterNode`* object is returned (by default).

Each *`ClusterNode`* object has a `left`, `right`, `dist`, `id`, and `count` attribute. The left and right attributes point to ClusterNode objects that were combined to generate the cluster. If both are None then the *`ClusterNode`* object is a leaf node, its count must be 1, and its distance is meaningless but set to 0.

Note: This function is provided for the convenience of the library user. ClusterNodes are not used as input to any of the functions in this library.

Parameters **Z** : ndarray
The linkage matrix in proper form (see the *`linkage`* function documentation).

rd : bool, optional
When False (default), a reference to the root *`ClusterNode`* object is returned. Otherwise, a tuple `(r, d)` is returned. `r` is a reference to the root node while `d` is a list of *`ClusterNode`* objects - one per original entry in the linkage matrix plus entries for all clustering steps. If a cluster id is less than the number of samples `n` in the data that the linkage matrix describes, then it corresponds to a singleton cluster (leaf node). See *`linkage`* for more information on the assignment of cluster ids to clusters.

Returns **tree** : ClusterNode or tuple (ClusterNode, list of ClusterNode)
If `rd` is False, a *`ClusterNode`*. If `rd` is True, a list of length `2*n - 1`, with `n` the number of samples. See the description of *rd* above for more details.

See also:

`linkage`, *`is_valid_linkage`*, *`ClusterNode`*

Examples

```
>>> from scipy.cluster import hierarchy
>>> x = np.random.rand(10).reshape(5, 2)
>>> Z = hierarchy.linkage(x)
>>> hierarchy.to_tree(Z)
<scipy.cluster.hierarchy.ClusterNode object at ...
>>> rootnode, nodelist = hierarchy.to_tree(Z, rd=True)
>>> rootnode
<scipy.cluster.hierarchy.ClusterNode object at ...
>>> len(nodelist)
9
```

`scipy.cluster.hierarchy.cut_tree(Z, n_clusters=None, height=None)`

Given a linkage matrix Z, return the cut tree.

Parameters

Z : scipy.cluster.linkage array
The linkage matrix.

n_clusters : array_like, optional
Number of clusters in the tree at the cut point.

height : array_like, optional
The height at which to cut the tree. Only possible for ultrametric trees.

Returns

cutree : array
An array indicating group membership at each agglomeration step. I.e., for a full cut tree, in the first column each data point is in its own cluster. At the next step, two nodes are merged. Finally all singleton and non-singleton clusters are in one group. If *n_clusters* or *height* is given, the columns correspond to the columns of *n_clusters* or *height*.

Examples

```
>>> from scipy import cluster
>>> np.random.seed(23)
>>> X = np.random.randn(50, 4)
>>> Z = cluster.hierarchy.ward(X)
>>> cutree = cluster.hierarchy.cut_tree(Z, n_clusters=[5, 10])
>>> cutree[:10]
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3],
       [3, 4],
       [2, 2],
       [0, 0],
       [1, 5],
       [3, 6],
       [4, 7]])
```

These are predicates for checking the validity of linkage and inconsistency matrices as well as for checking isomorphism of two flat cluster assignments.

is_valid_im(R[, warning, throw, name])	Returns True if the inconsistency matrix passed is valid.
is_valid_linkage(Z[, warning, throw, name])	Checks the validity of a linkage matrix.
is_isomorphic(T1, T2)	Determines if two different cluster assignments are equivalent.
is_monotonic(Z)	Returns True if the linkage passed is monotonic.

Continued on next page

Table 5.8 – continued from previous page

correspond(Z, Y)	Checks for correspondence between linkage and condensed distance matrices
num_obs_linkage(Z)	Returns the number of original observations of the linkage matrix passed.

scipy.cluster.hierarchy.**is_valid_im**(*R*, *warning=False*, *throw=False*, *name=None*)

Returns True if the inconsistency matrix passed is valid.

It must be a n by 4 array of doubles. The standard deviations `R[:,1]` must be nonnegative. The link counts `R[:,2]` must be positive and no greater than $n-1$.

Parameters

R : ndarray
: The inconsistency matrix to check for validity.

warning : bool, optional
: When True, issues a Python warning if the linkage matrix passed is invalid.

throw : bool, optional
: When True, throws a Python exception if the linkage matrix passed is invalid.

name : str, optional
: This string refers to the variable name of the invalid linkage matrix.

Returns

b : bool
: True if the inconsistency matrix is valid.

scipy.cluster.hierarchy.**is_valid_linkage**(*Z*, *warning=False*, *throw=False*, *name=None*)

Checks the validity of a linkage matrix.

A linkage matrix is valid if it is a two dimensional array (type double) with n rows and 4 columns. The first two columns must contain indices between 0 and $2n-1$. For a given row `i`, the following two expressions have to hold:

$$0 \leq \mathtt{Z}[\mathtt{i},0] \leq i+n-10 \leq Z[i,1] \leq i+n-1$$

I.e. a cluster cannot join another cluster unless the cluster being joined has been generated.

Parameters

Z : array_like
: Linkage matrix.

warning : bool, optional
: When True, issues a Python warning if the linkage matrix passed is invalid.

throw : bool, optional
: When True, throws a Python exception if the linkage matrix passed is invalid.

name : str, optional
: This string refers to the variable name of the invalid linkage matrix.

Returns

b : bool
: True if the inconsistency matrix is valid.

scipy.cluster.hierarchy.**is_isomorphic**(*T1*, *T2*)

Determines if two different cluster assignments are equivalent.

Parameters

T1 : array_like
: An assignment of singleton cluster ids to flat cluster ids.

T2 : array_like
: An assignment of singleton cluster ids to flat cluster ids.

Returns

b : bool
: Whether the flat cluster assignments *T1* and *T2* are equivalent.

`scipy.cluster.hierarchy.`**`is_monotonic`**`(Z)`

Returns True if the linkage passed is monotonic.

The linkage is monotonic if for every cluster s and t joined, the distance between them is no less than the distance between any previously joined clusters.

Parameters **Z** : ndarray
The linkage matrix to check for monotonicity.

Returns **b** : bool
A boolean indicating whether the linkage is monotonic.

`scipy.cluster.hierarchy.`**`correspond`**`(Z, Y)`

Checks for correspondence between linkage and condensed distance matrices

They must have the same number of original observations for the check to succeed.

This function is useful as a sanity check in algorithms that make extensive use of linkage and distance matrices that must correspond to the same set of original observations.

Parameters **Z** : array_like
The linkage matrix to check for correspondence.

Y : array_like
The condensed distance matrix to check for correspondence.

Returns **b** : bool
A boolean indicating whether the linkage matrix and distance matrix could possibly correspond to one another.

`scipy.cluster.hierarchy.`**`num_obs_linkage`**`(Z)`

Returns the number of original observations of the linkage matrix passed.

Parameters **Z** : ndarray
The linkage matrix on which to perform the operation.

Returns **n** : int
The number of original observations in the linkage.

Utility routines for plotting:

set_link_color_palette(palette)	Set list of matplotlib color codes for use by dendrogram.

`scipy.cluster.hierarchy.`**`set_link_color_palette`**`(`*palette*`)`

Set list of matplotlib color codes for use by dendrogram.

Note that this palette is global (i.e. setting it once changes the colors for all subsequent calls to *dendrogram*) and that it affects only the the colors below `color_threshold`.

Note that *dendrogram* also accepts a custom coloring function through its `link_color_func` keyword, which is more flexible and non-global.

Parameters **palette** : list of str or None
A list of matplotlib color codes. The order of the color codes is the order in which the colors are cycled through when color thresholding in the dendrogram.
If `None`, resets the palette to its default (which is `['g', 'r', 'c', 'm', 'y', 'k']`).

Returns None

See also:

dendrogram

Notes

Ability to reset the palette with `None` added in Scipy 0.17.0.

Examples

```
>>> from scipy.cluster import hierarchy
>>> ytdist = np.array([662., 877., 255., 412., 996., 295., 468., 268., 400.,
...                    754., 564., 138., 219., 869., 669.])
>>> Z = hierarchy.linkage(ytdist, 'single')
>>> dn = hierarchy.dendrogram(Z, no_plot=True)
>>> dn['color_list']
['g', 'b', 'b', 'b', 'b']
>>> hierarchy.set_link_color_palette(['c', 'm', 'y', 'k'])
>>> dn = hierarchy.dendrogram(Z, no_plot=True)
>>> dn['color_list']
['c', 'b', 'b', 'b', 'b']
>>> dn = hierarchy.dendrogram(Z, no_plot=True, color_threshold=267,
...                           above_threshold_color='k')
>>> dn['color_list']
['c', 'm', 'm', 'k', 'k']
```

Now reset the color palette to its default:

```
>>> hierarchy.set_link_color_palette(None)
```

5.3.1 References

- MATLAB and MathWorks are registered trademarks of The MathWorks, Inc.
- Mathematica is a registered trademark of The Wolfram Research, Inc.

5.4 Constants (`scipy.constants`)

Physical and mathematical constants and units.

5.4.1 Mathematical constants

`pi`	Pi
`golden`	Golden ratio
`golden_ratio`	Golden ratio

5.4.2 Physical constants

c	speed of light in vacuum
speed_of_light	speed of light in vacuum
mu_0	the magnetic constant μ_0
epsilon_0	the electric constant (vacuum permittivity), ϵ_0
h	the Planck constant h
Planck	the Planck constant h
hbar	$\hbar = h/(2\pi)$
G	Newtonian constant of gravitation
gravitational_constant	Newtonian constant of gravitation
g	standard acceleration of gravity
e	elementary charge
elementary_charge	elementary charge
R	molar gas constant
gas_constant	molar gas constant
alpha	fine-structure constant
fine_structure	fine-structure constant
N_A	Avogadro constant
Avogadro	Avogadro constant
k	Boltzmann constant
Boltzmann	Boltzmann constant
sigma	Stefan-Boltzmann constant σ
Stefan_Boltzmann	Stefan-Boltzmann constant σ
Wien	Wien displacement law constant
Rydberg	Rydberg constant
m_e	electron mass
electron_mass	electron mass
m_p	proton mass
proton_mass	proton mass
m_n	neutron mass
neutron_mass	neutron mass

Constants database

In addition to the above variables, *scipy.constants* also contains the 2014 CODATA recommended values *[CODATA2014]* database containing more physical constants.

value(key)	Value in physical_constants indexed by key
unit(key)	Unit in physical_constants indexed by key
precision(key)	Relative precision in physical_constants indexed by key
find([sub, disp])	Return list of physical_constant keys containing a given string.
ConstantWarning	Accessing a constant no longer in current CODATA data set

scipy.constants.**value**(*key*)

Value in physical_constants indexed by key

Parameters **key** : Python string or unicode

Key in dictionary *physical_constants*

Returns **value** : float

Value in *physical_constants* corresponding to *key*

See also:

codata Contains the description of *physical_constants*, which, as a dictionary literal object, does not itself possess a docstring.

Examples

```
>>> from scipy import constants
>>> constants.value(u'elementary charge')
    1.6021766208e-19
```

`scipy.constants.unit(key)`
Unit in physical_constants indexed by key

Parameters **key** : Python string or unicode
Key in dictionary *physical_constants*

Returns **unit** : Python string
Unit in *physical_constants* corresponding to *key*

See also:

codata Contains the description of *physical_constants*, which, as a dictionary literal object, does not itself possess a docstring.

Examples

```
>>> from scipy import constants
>>> constants.unit(u'proton mass')
'kg'
```

`scipy.constants.precision(key)`
Relative precision in physical_constants indexed by key

Parameters **key** : Python string or unicode
Key in dictionary *physical_constants*

Returns **prec** : float
Relative precision in *physical_constants* corresponding to *key*

See also:

codata Contains the description of *physical_constants*, which, as a dictionary literal object, does not itself possess a docstring.

Examples

```
>>> from scipy import constants
>>> constants.precision(u'proton mass')
1.2555138746605121e-08
```

`scipy.constants.find(sub=None, disp=False)`
Return list of physical_constant keys containing a given string.

Parameters **sub** : str, unicode
Sub-string to search keys for. By default, return all keys.

disp : bool

If True, print the keys that are found, and return None. Otherwise, return the list of keys without printing anything.

Returns **keys** : list or None
If *disp* is False, the list of keys is returned. Otherwise, None is returned.

See also:

codata Contains the description of *physical_constants*, which, as a dictionary literal object, does not itself possess a docstring.

exception scipy.constants.**ConstantWarning**
Accessing a constant no longer in current CODATA data set

scipy.constants.**physical_constants**
Dictionary of physical constants, of the format `physical_constants[name] = (value, unit, uncertainty)`.

Available constants:

alpha particle mass	6.64465723e-27 kg
alpha particle mass energy equivalent	5.971920097e-10 J
alpha particle mass energy equivalent in MeV	3727.379378 MeV
alpha particle mass in u	4.00150617913 u
alpha particle molar mass	0.00400150617913 kg mol^-1
alpha particle-electron mass ratio	7294.29954136
alpha particle-proton mass ratio	3.97259968907
Angstrom star	1.00001495e-10 m
atomic mass constant	1.66053904e-27 kg
atomic mass constant energy equivalent	1.492418062e-10 J
atomic mass constant energy equivalent in MeV	931.4940954 MeV
atomic mass unit-electron volt relationship	931494095.4 eV
atomic mass unit-hartree relationship	34231776.902 E_h
atomic mass unit-hertz relationship	2.2523427206e+23 Hz
atomic mass unit-inverse meter relationship	7.5130066166e+14 m^-1
atomic mass unit-joule relationship	1.492418062e-10 J
atomic mass unit-kelvin relationship	1.08095438e+13 K
atomic mass unit-kilogram relationship	1.66053904e-27 kg
atomic unit of 1st hyperpolarizability	3.206361329e-53 C^3 m^3 J^-2
atomic unit of 2nd hyperpolarizability	6.235380085e-65 C^4 m^4 J^-3
atomic unit of action	1.0545718e-34 J s
atomic unit of charge	1.6021766208e-19 C
atomic unit of charge density	1.081202377e+12 C m^-3
atomic unit of current	0.006623618183 A
atomic unit of electric dipole mom.	8.478353552e-30 C m
atomic unit of electric field	5.142206707e+11 V m^-1
atomic unit of electric field gradient	9.717362356e+21 V m^-2
atomic unit of electric polarizability	1.6487772731e-41 C^2 m^2 J^-1
atomic unit of electric potential	27.21138602 V
atomic unit of electric quadrupole mom.	4.486551484e-40 C m^2
atomic unit of energy	4.35974465e-18 J
atomic unit of force	8.23872336e-08 N
atomic unit of length	5.2917721067e-11 m
atomic unit of mag. dipole mom.	1.854801999e-23 J T^-1

Continued on next page

Table 5.11 – continued from previous page

atomic unit of mag. flux density	235051.755 T
atomic unit of magnetizability	7.8910365886e-29 J T^-2
atomic unit of mass	9.10938356e-31 kg
atomic unit of mom.um	1.992851882e-24 kg m s^-1
atomic unit of permittivity	1.11265005605e-10 F m^-1
atomic unit of time	2.41888432651e-17 s
atomic unit of velocity	2187691.26277 m s^-1
Avogadro constant	6.022140857e+23 mol^-1
Bohr magneton	9.274009994e-24 J T^-1
Bohr magneton in eV/T	5.7883818012e-05 eV T^-1
Bohr magneton in Hz/T	13996245042.0 Hz T^-1
Bohr magneton in inverse meters per tesla	46.68644814 m^-1 T^-1
Bohr magneton in K/T	0.67171405 K T^-1
Bohr radius	5.2917721067e-11 m
Boltzmann constant	1.38064852e-23 J K^-1
Boltzmann constant in eV/K	8.6173303e-05 eV K^-1
Boltzmann constant in Hz/K	20836612000.0 Hz K^-1
Boltzmann constant in inverse meters per kelvin	69.503457 m^-1 K^-1
characteristic impedance of vacuum	376.730313462 ohm
classical electron radius	2.8179403227e-15 m
Compton wavelength	2.4263102367e-12 m
Compton wavelength over 2 pi	3.8615926764e-13 m
conductance quantum	7.748091731e-05 S
conventional value of Josephson constant	4.835979e+14 Hz V^-1
conventional value of von Klitzing constant	25812.807 ohm
Cu x unit	1.00207697e-13 m
deuteron g factor	0.8574382311
deuteron mag. mom.	4.33073504e-27 J T^-1
deuteron mag. mom. to Bohr magneton ratio	0.0004669754554
deuteron mag. mom. to nuclear magneton ratio	0.8574382311
deuteron mass	3.343583719e-27 kg
deuteron mass energy equivalent	3.005063183e-10 J
deuteron mass energy equivalent in MeV	1875.612928 MeV
deuteron mass in u	2.01355321275 u
deuteron molar mass	0.00201355321274 kg mol^-1
deuteron rms charge radius	2.1413e-15 m
deuteron-electron mag. mom. ratio	-0.0004664345535
deuteron-electron mass ratio	3670.48296785
deuteron-neutron mag. mom. ratio	-0.44820652
deuteron-proton mag. mom. ratio	0.3070122077
deuteron-proton mass ratio	1.99900750087
electric constant	8.85418781762e-12 F m^-1
electron charge to mass quotient	-1.758820024e+11 C kg^-1
electron g factor	-2.00231930436
electron gyromag. ratio	1.760859644e+11 s^-1 T^-1
electron gyromag. ratio over 2 pi	28024.95164 MHz T^-1
electron mag. mom.	-9.28476462e-24 J T^-1
electron mag. mom. anomaly	0.00115965218091
electron mag. mom. to Bohr magneton ratio	-1.00115965218
electron mag. mom. to nuclear magneton ratio	-1838.28197234

Continued on next page

Table 5.11 – continued from previous page

electron mass	9.10938356e-31 kg
electron mass energy equivalent	8.18710565e-14 J
electron mass energy equivalent in MeV	0.5109989461 MeV
electron mass in u	0.00054857990907 u
electron molar mass	5.4857990907e-07 kg mol^-1
electron to alpha particle mass ratio	0.00013709335548
electron to shielded helion mag. mom. ratio	864.058257
electron to shielded proton mag. mom. ratio	-658.2275971
electron volt	1.6021766208e-19 J
electron volt-atomic mass unit relationship	1.0735441105e-09 u
electron volt-hartree relationship	0.03674932248 E_h
electron volt-hertz relationship	2.417989262e+14 Hz
electron volt-inverse meter relationship	806554.4005 m^-1
electron volt-joule relationship	1.6021766208e-19 J
electron volt-kelvin relationship	11604.5221 K
electron volt-kilogram relationship	1.782661907e-36 kg
electron-deuteron mag. mom. ratio	-2143.923499
electron-deuteron mass ratio	0.000272443710748
electron-helion mass ratio	0.000181954307485
electron-muon mag. mom. ratio	206.766988
electron-muon mass ratio	0.0048363317
electron-neutron mag. mom. ratio	960.9205
electron-neutron mass ratio	0.00054386734428
electron-proton mag. mom. ratio	-658.2106866
electron-proton mass ratio	0.000544617021352
electron-tau mass ratio	0.000287592
electron-triton mass ratio	0.00018192000622
elementary charge	1.6021766208e-19 C
elementary charge over h	2.417989262e+14 A J^-1
Faraday constant	96485.33289 C mol^-1
Faraday constant for conventional electric current	96485.3251 C_90 mol^-1
Fermi coupling constant	1.1663787e-05 GeV^-2
fine-structure constant	0.0072973525664
first radiation constant	3.74177179e-16 W m^2
first radiation constant for spectral radiance	1.191042953e-16 W m^2 sr^-1
Hartree energy	4.35974465e-18 J
Hartree energy in eV	27.21138602 eV
hartree-atomic mass unit relationship	2.9212623197e-08 u
hartree-electron volt relationship	27.21138602 eV
hartree-hertz relationship	6.57968392071e+15 Hz
hartree-inverse meter relationship	21947463.137 m^-1
hartree-joule relationship	4.35974465e-18 J
hartree-kelvin relationship	315775.13 K
hartree-kilogram relationship	4.850870129e-35 kg
helion g factor	-4.255250616
helion mag. mom.	-1.074617522e-26 J T^-1
helion mag. mom. to Bohr magneton ratio	-0.001158740958
helion mag. mom. to nuclear magneton ratio	-2.127625308
helion mass	5.0064127e-27 kg
helion mass energy equivalent	4.499539341e-10 J

Continued on next page

Table 5.11 – continued from previous page

helion mass energy equivalent in MeV	2808.391586 MeV
helion mass in u	3.01493224673 u
helion molar mass	0.00301493224673 kg mol^-1
helion-electron mass ratio	5495.88527922
helion-proton mass ratio	2.99315267046
hertz-atomic mass unit relationship	4.4398216616e-24 u
hertz-electron volt relationship	4.135667662e-15 eV
hertz-hartree relationship	1.51982984601e-16 E_h
hertz-inverse meter relationship	3.33564095198e-09 m^-1
hertz-joule relationship	6.62607004e-34 J
hertz-kelvin relationship	4.7992447e-11 K
hertz-kilogram relationship	7.372497201e-51 kg
inverse fine-structure constant	137.035999139
inverse meter-atomic mass unit relationship	1.331025049e-15 u
inverse meter-electron volt relationship	1.2398419739e-06 eV
inverse meter-hartree relationship	4.55633525277e-08 E_h
inverse meter-hertz relationship	299792458.0 Hz
inverse meter-joule relationship	1.986445824e-25 J
inverse meter-kelvin relationship	0.0143877736 K
inverse meter-kilogram relationship	2.210219057e-42 kg
inverse of conductance quantum	12906.4037278 ohm
Josephson constant	4.835978525e+14 Hz V^-1
joule-atomic mass unit relationship	6700535363.0 u
joule-electron volt relationship	6.241509126e+18 eV
joule-hartree relationship	2.293712317e+17 E_h
joule-hertz relationship	1.509190205e+33 Hz
joule-inverse meter relationship	5.034116651e+24 m^-1
joule-kelvin relationship	7.2429731e+22 K
joule-kilogram relationship	1.11265005605e-17 kg
kelvin-atomic mass unit relationship	9.2510842e-14 u
kelvin-electron volt relationship	8.6173303e-05 eV
kelvin-hartree relationship	3.1668105e-06 E_h
kelvin-hertz relationship	20836612000.0 Hz
kelvin-inverse meter relationship	69.503457 m^-1
kelvin-joule relationship	1.38064852e-23 J
kelvin-kilogram relationship	1.53617865e-40 kg
kilogram-atomic mass unit relationship	6.022140857e+26 u
kilogram-electron volt relationship	5.60958865e+35 eV
kilogram-hartree relationship	2.061485823e+34 E_h
kilogram-hertz relationship	1.356392512e+50 Hz
kilogram-inverse meter relationship	4.524438411e+41 m^-1
kilogram-joule relationship	8.98755178737e+16 J
kilogram-kelvin relationship	6.5096595e+39 K
lattice parameter of silicon	5.431020504e-10 m
Loschmidt constant (273.15 K, 100 kPa)	2.6516467e+25 m^-3
Loschmidt constant (273.15 K, 101.325 kPa)	2.6867811e+25 m^-3
mag. constant	1.25663706144e-06 N A^-2
mag. flux quantum	2.067833831e-15 Wb
Mo x unit	1.00209952e-13 m
molar gas constant	8.3144598 J mol^-1 K^-1

Continued on next page

Table 5.11 – continued from previous page

molar mass constant	0.001 kg mol^-1
molar mass of carbon-12	0.012 kg mol^-1
molar Planck constant	3.990312711e-10 J s mol^-1
molar Planck constant times c	0.119626565582 J m mol^-1
molar volume of ideal gas (273.15 K, 100 kPa)	0.022710947 m^3 mol^-1
molar volume of ideal gas (273.15 K, 101.325 kPa)	0.022413962 m^3 mol^-1
molar volume of silicon	1.205883214e-05 m^3 mol^-1
muon Compton wavelength	1.173444111e-14 m
muon Compton wavelength over 2 pi	1.867594308e-15 m
muon g factor	-2.0023318418
muon mag. mom.	-4.49044826e-26 J T^-1
muon mag. mom. anomaly	0.00116592089
muon mag. mom. to Bohr magneton ratio	-0.00484197048
muon mag. mom. to nuclear magneton ratio	-8.89059705
muon mass	1.883531594e-28 kg
muon mass energy equivalent	1.692833774e-11 J
muon mass energy equivalent in MeV	105.6583745 MeV
muon mass in u	0.1134289257 u
muon molar mass	0.0001134289257 kg mol^-1
muon-electron mass ratio	206.7682826
muon-neutron mass ratio	0.1124545167
muon-proton mag. mom. ratio	-3.183345142
muon-proton mass ratio	0.1126095262
muon-tau mass ratio	0.0594649
natural unit of action	1.0545718e-34 J s
natural unit of action in eV s	6.582119514e-16 eV s
natural unit of energy	8.18710565e-14 J
natural unit of energy in MeV	0.5109989461 MeV
natural unit of length	3.8615926764e-13 m
natural unit of mass	9.10938356e-31 kg
natural unit of mom.um	2.730924488e-22 kg m s^-1
natural unit of mom.um in MeV/c	0.5109989461 MeV/c
natural unit of time	1.28808866712e-21 s
natural unit of velocity	299792458.0 m s^-1
neutron Compton wavelength	1.31959090481e-15 m
neutron Compton wavelength over 2 pi	2.1001941536e-16 m
neutron g factor	-3.82608545
neutron gyromag. ratio	183247172.0 s^-1 T^-1
neutron gyromag. ratio over 2 pi	29.1646933 MHz T^-1
neutron mag. mom.	-9.662365e-27 J T^-1
neutron mag. mom. to Bohr magneton ratio	-0.00104187563
neutron mag. mom. to nuclear magneton ratio	-1.91304273
neutron mass	1.674927471e-27 kg
neutron mass energy equivalent	1.505349739e-10 J
neutron mass energy equivalent in MeV	939.5654133 MeV
neutron mass in u	1.00866491588 u
neutron molar mass	0.00100866491588 kg mol^-1
neutron to shielded proton mag. mom. ratio	-0.68499694
neutron-electron mag. mom. ratio	0.00104066882
neutron-electron mass ratio	1838.68366158

Continued on next page

Table 5.11 – continued from previous page

neutron-muon mass ratio	8.89248408
neutron-proton mag. mom. ratio	-0.68497934
neutron-proton mass difference	2.30557377e-30
neutron-proton mass difference energy equivalent	2.07214637e-13
neutron-proton mass difference energy equivalent in MeV	1.29333205
neutron-proton mass difference in u	0.001388449
neutron-proton mass ratio	1.00137841898
neutron-tau mass ratio	0.52879
Newtonian constant of gravitation	6.67408e-11 m^3 kg^-1 s^-2
Newtonian constant of gravitation over h-bar c	6.70861e-39 (GeV/c^2)^-2
nuclear magneton	5.050783699e-27 J T^-1
nuclear magneton in eV/T	3.152451255e-08 eV T^-1
nuclear magneton in inverse meters per tesla	0.02542623432 m^-1 T^-1
nuclear magneton in K/T	0.0003658269 K T^-1
nuclear magneton in MHz/T	7.622593285 MHz T^-1
Planck constant	6.62607004e-34 J s
Planck constant in eV s	4.135667662e-15 eV s
Planck constant over 2 pi	1.0545718e-34 J s
Planck constant over 2 pi in eV s	6.582119514e-16 eV s
Planck constant over 2 pi times c in MeV fm	197.3269788 MeV fm
Planck length	1.616229e-35 m
Planck mass	2.17647e-08 kg
Planck mass energy equivalent in GeV	1.22091e+19 GeV
Planck temperature	1.416808e+32 K
Planck time	5.39116e-44 s
proton charge to mass quotient	95788332.26 C kg^-1
proton Compton wavelength	1.32140985396e-15 m
proton Compton wavelength over 2 pi	2.10308910109e-16 m
proton g factor	5.585694702
proton gyromag. ratio	267522190.0 s^-1 T^-1
proton gyromag. ratio over 2 pi	42.57747892 MHz T^-1
proton mag. mom.	1.4106067873e-26 J T^-1
proton mag. mom. to Bohr magneton ratio	0.0015210322053
proton mag. mom. to nuclear magneton ratio	2.7928473508
proton mag. shielding correction	2.5691e-05
proton mass	1.672621898e-27 kg
proton mass energy equivalent	1.503277593e-10 J
proton mass energy equivalent in MeV	938.2720813 MeV
proton mass in u	1.00727646688 u
proton molar mass	0.00100727646688 kg mol^-1
proton rms charge radius	8.751e-16 m
proton-electron mass ratio	1836.15267389
proton-muon mass ratio	8.88024338
proton-neutron mag. mom. ratio	-1.45989805
proton-neutron mass ratio	0.99862347844
proton-tau mass ratio	0.528063
quantum of circulation	0.00036369475486 m^2 s^-1
quantum of circulation times 2	0.00072738950972 m^2 s^-1
Rydberg constant	10973731.5685 m^-1
Rydberg constant times c in Hz	3.28984196036e+15 Hz

Continued on next page

Table 5.11 – continued from previous page

Rydberg constant times hc in eV	13.605693009 eV
Rydberg constant times hc in J	2.179872325e-18 J
Sackur-Tetrode constant (1 K, 100 kPa)	-1.1517084
Sackur-Tetrode constant (1 K, 101.325 kPa)	-1.1648714
second radiation constant	0.0143877736 m K
shielded helion gyromag. ratio	203789458.5 s^-1 T^-1
shielded helion gyromag. ratio over 2 pi	32.43409966 MHz T^-1
shielded helion mag. mom.	-1.07455308e-26 J T^-1
shielded helion mag. mom. to Bohr magneton ratio	-0.001158671471
shielded helion mag. mom. to nuclear magneton ratio	-2.12749772
shielded helion to proton mag. mom. ratio	-0.7617665603
shielded helion to shielded proton mag. mom. ratio	-0.7617861313
shielded proton gyromag. ratio	267515317.1 s^-1 T^-1
shielded proton gyromag. ratio over 2 pi	42.57638507 MHz T^-1
shielded proton mag. mom.	1.410570547e-26 J T^-1
shielded proton mag. mom. to Bohr magneton ratio	0.001520993128
shielded proton mag. mom. to nuclear magneton ratio	2.7927756
speed of light in vacuum	299792458.0 m s^-1
standard acceleration of gravity	9.80665 m s^-2
standard atmosphere	101325.0 Pa
standard-state pressure	100000.0 Pa
Stefan-Boltzmann constant	5.670367e-08 W m^-2 K^-4
tau Compton wavelength	6.97787e-16 m
tau Compton wavelength over 2 pi	1.11056e-16 m
tau mass	3.16747e-27 kg
tau mass energy equivalent	2.84678e-10 J
tau mass energy equivalent in MeV	1776.82 MeV
tau mass in u	1.90749 u
tau molar mass	0.00190749 kg mol^-1
tau-electron mass ratio	3477.15
tau-muon mass ratio	16.8167
tau-neutron mass ratio	1.89111
tau-proton mass ratio	1.89372
Thomson cross section	6.6524587158e-29 m^2
triton g factor	5.95792492
triton mag. mom.	1.504609503e-26 J T^-1
triton mag. mom. to Bohr magneton ratio	0.0016223936616
triton mag. mom. to nuclear magneton ratio	2.97896246
triton mass	5.007356665e-27 kg
triton mass energy equivalent	4.500387735e-10 J
triton mass energy equivalent in MeV	2808.921112 MeV
triton mass in u	3.01550071632 u
triton molar mass	0.00301550071632 kg mol^-1
triton-electron mass ratio	5496.92153588
triton-proton mass ratio	2.99371703348
unified atomic mass unit	1.66053904e-27 kg
von Klitzing constant	25812.8074555 ohm
weak mixing angle	0.2223
Wien frequency displacement law constant	58789238000.0 Hz K^-1
Wien wavelength displacement law constant	0.0028977729 m K

Continued on next page

Table 5.11 – continued from previous page

{220} lattice spacing of silicon	1.920155714e-10 m

5.4.3 Units

SI prefixes

yotta	10^{24}
zetta	10^{21}
exa	10^{18}
peta	10^{15}
tera	10^{12}
giga	10^{9}
mega	10^{6}
kilo	10^{3}
hecto	10^{2}
deka	10^{1}
deci	10^{-1}
centi	10^{-2}
milli	10^{-3}
micro	10^{-6}
nano	10^{-9}
pico	10^{-12}
femto	10^{-15}
atto	10^{-18}
zepto	10^{-21}

Binary prefixes

kibi	2^{10}
mebi	2^{20}
gibi	2^{30}
tebi	2^{40}
pebi	2^{50}
exbi	2^{60}
zebi	2^{70}
yobi	2^{80}

Weight

gram	10^{-3} kg
metric_ton	10^{3} kg
grain	one grain in kg
lb	one pound (avoirdupous) in kg
pound	one pound (avoirdupous) in kg
oz	one ounce in kg
ounce	one ounce in kg
stone	one stone in kg
grain	one grain in kg
long_ton	one long ton in kg
short_ton	one short ton in kg
troy_ounce	one Troy ounce in kg
troy_pound	one Troy pound in kg
carat	one carat in kg
m_u	atomic mass constant (in kg)
u	atomic mass constant (in kg)
atomic_mass	atomic mass constant (in kg)

Angle

degree	degree in radians
arcmin	arc minute in radians
arcminute	arc minute in radians
arcsec	arc second in radians
arcsecond	arc second in radians

Time

minute	one minute in seconds
hour	one hour in seconds
day	one day in seconds
week	one week in seconds
year	one year (365 days) in seconds
Julian_year	one Julian year (365.25 days) in seconds

Length

`inch`	one inch in meters
`foot`	one foot in meters
`yard`	one yard in meters
`mile`	one mile in meters
`mil`	one mil in meters
`pt`	one point in meters
`point`	one point in meters
`survey_foot`	one survey foot in meters
`survey_mile`	one survey mile in meters
`nautical_mile`	one nautical mile in meters
`fermi`	one Fermi in meters
`angstrom`	one Angstrom in meters
`micron`	one micron in meters
`au`	one astronomical unit in meters
`astronomical_unit`	one astronomical unit in meters
`light_year`	one light year in meters
`parsec`	one parsec in meters

Pressure

`atm`	standard atmosphere in pascals
`atmosphere`	standard atmosphere in pascals
`bar`	one bar in pascals
`torr`	one torr (mmHg) in pascals
`mmHg`	one torr (mmHg) in pascals
`psi`	one psi in pascals

Area

`hectare`	one hectare in square meters
`acre`	one acre in square meters

Volume

`liter`	one liter in cubic meters
`litre`	one liter in cubic meters
`gallon`	one gallon (US) in cubic meters
`gallon_US`	one gallon (US) in cubic meters
`gallon_imp`	one gallon (UK) in cubic meters
`fluid_ounce`	one fluid ounce (US) in cubic meters
`fluid_ounce_US`	one fluid ounce (US) in cubic meters
`fluid_ounce_imp`	one fluid ounce (UK) in cubic meters
`bbl`	one barrel in cubic meters
`barrel`	one barrel in cubic meters

Speed

kmh	kilometers per hour in meters per second
mph	miles per hour in meters per second
mach	one Mach (approx., at 15 C, 1 atm) in meters per second
speed_of_sound	one Mach (approx., at 15 C, 1 atm) in meters per second
knot	one knot in meters per second

Temperature

zero_Celsius	zero of Celsius scale in Kelvin
degree_Fahrenheit	one Fahrenheit (only differences) in Kelvins

convert_temperature(val, old_scale, new_scale)	Convert from a temperature scale to another one among Celsius, Kelvin, Fahrenheit and Rankine scales.
C2K(*args, **kwds)	*C2K* is deprecated!
K2C(*args, **kwds)	*K2C* is deprecated!
F2C(*args, **kwds)	*F2C* is deprecated!
C2F(*args, **kwds)	*C2F* is deprecated!
F2K(*args, **kwds)	*F2K* is deprecated!
K2F(*args, **kwds)	*K2F* is deprecated!

scipy.constants.**convert_temperature**(*val*, *old_scale*, *new_scale*)

Convert from a temperature scale to another one among Celsius, Kelvin, Fahrenheit and Rankine scales.

Parameters

val : array_like

Value(s) of the temperature(s) to be converted expressed in the original scale.

old_scale: str

Specifies as a string the original scale from which the temperature value(s) will be converted. Supported scales are Celsius ('Celsius', 'celsius', 'C' or 'c'), Kelvin ('Kelvin', 'kelvin', 'K', 'k'), Fahrenheit ('Fahrenheit', 'fahrenheit', 'F' or 'f') and Rankine ('Rankine', 'rankine', 'R', 'r').

new_scale: str

Specifies as a string the new scale to which the temperature value(s) will be converted. Supported scales are Celsius ('Celsius', 'celsius', 'C' or 'c'), Kelvin ('Kelvin', 'kelvin', 'K', 'k'), Fahrenheit ('Fahrenheit', 'fahrenheit', 'F' or 'f') and Rankine ('Rankine', 'rankine', 'R', 'r').

Returns

res : float or array of floats

Value(s) of the converted temperature(s) expressed in the new scale.

Notes

New in version 0.18.0.

Examples

```
>>> from scipy.constants import convert_temperature
>>> convert_temperature(np.array([-40, 40.0]), 'Celsius', 'Kelvin')
array([ 233.15,  313.15])
```

scipy.constants.**C2K**(**args*, ***kwds*)

C2K is deprecated! scipy.constants.C2K is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Celsius to Kelvin

Parameters **C** : array_like
Celsius temperature(s) to be converted.

Returns **K** : float or array of floats
Equivalent Kelvin temperature(s).

Notes

Computes `K = C + zero_Celsius` where `zero_Celsius` = 273.15, i.e., (the absolute value of) temperature "absolute zero" as measured in Celsius.

Examples

```
>>> from scipy.constants import C2K
>>> C2K(np.array([-40, 40.0]))
array([ 233.15,  313.15])
```

`scipy.constants.`**`K2C`**`(*args, **kwds)`

K2C is deprecated! scipy.constants.K2C is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Kelvin to Celsius

Parameters **K** : array_like
Kelvin temperature(s) to be converted.

Returns **C** : float or array of floats
Equivalent Celsius temperature(s).

Notes

Computes `C = K - zero_Celsius` where `zero_Celsius` = 273.15, i.e., (the absolute value of) temperature "absolute zero" as measured in Celsius.

Examples

```
>>> from scipy.constants import K2C
>>> K2C(np.array([233.15, 313.15]))
array([-40.,  40.])
```

`scipy.constants.`**`F2C`**`(*args, **kwds)`

F2C is deprecated! scipy.constants.F2C is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Fahrenheit to Celsius

Parameters **F** : array_like
Fahrenheit temperature(s) to be converted.

Returns **C** : float or array of floats
Equivalent Celsius temperature(s).

Notes

Computes `C = (F - 32) / 1.8`.

Examples

```
>>> from scipy.constants import F2C
>>> F2C(np.array([-40, 40.0]))
array([-40.        ,   4.44444444])
```

`scipy.constants.`**`C2F`**(**args*, ***kwds*)

C2F is deprecated! scipy.constants.C2F is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Celsius to Fahrenheit

Parameters **C** : array_like
Celsius temperature(s) to be converted.

Returns **F** : float or array of floats
Equivalent Fahrenheit temperature(s).

Notes

Computes `F = 1.8 * C + 32`.

Examples

```
>>> from scipy.constants import C2F
>>> C2F(np.array([-40, 40.0]))
array([ -40.,  104.])
```

`scipy.constants.`**`F2K`**(**args*, ***kwds*)

F2K is deprecated! scipy.constants.F2K is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Fahrenheit to Kelvin

Parameters **F** : array_like
Fahrenheit temperature(s) to be converted.

Returns **K** : float or array of floats
Equivalent Kelvin temperature(s).

Notes

Computes `K = (F - 32)/1.8 + zero_Celsius` where `zero_Celsius` = 273.15, i.e., (the absolute value of) temperature "absolute zero" as measured in Celsius.

Examples

```
>>> from scipy.constants import F2K
>>> F2K(np.array([-40, 104]))
array([ 233.15,  313.15])
```

`scipy.constants.`**`K2F`**(**args*, ***kwds*)

K2F is deprecated! scipy.constants.K2F is deprecated in scipy 0.18.0. Use scipy.constants.convert_temperature instead. Note that the new function has a different signature.

Convert Kelvin to Fahrenheit

Parameters **K** : array_like
Kelvin temperature(s) to be converted.

Returns **F** : float or array of floats

Equivalent Fahrenheit temperature(s).

Notes

Computes `F = 1.8 * (K - zero_Celsius) + 32` where `zero_Celsius` = 273.15, i.e., (the absolute value of) temperature "absolute zero" as measured in Celsius.

Examples

```
>>> from scipy.constants import K2F
>>> K2F(np.array([233.15,  313.15]))
array([ -40.,  104.])
```

Energy

eV	one electron volt in Joules
electron_volt	one electron volt in Joules
calorie	one calorie (thermochemical) in Joules
calorie_th	one calorie (thermochemical) in Joules
calorie_IT	one calorie (International Steam Table calorie, 1956) in Joules
erg	one erg in Joules
Btu	one British thermal unit (International Steam Table) in Joules
Btu_IT	one British thermal unit (International Steam Table) in Joules
Btu_th	one British thermal unit (thermochemical) in Joules
ton_TNT	one ton of TNT in Joules

Power

hp	one horsepower in watts
horsepower	one horsepower in watts

Force

dyn	one dyne in newtons
dyne	one dyne in newtons
lbf	one pound force in newtons
pound_force	one pound force in newtons
kgf	one kilogram force in newtons
kilogram_force	one kilogram force in newtons

Optics

lambda2nu(lambda_)	Convert wavelength to optical frequency
nu2lambda(nu)	Convert optical frequency to wavelength.

`scipy.constants.lambda2nu(lambda_)`
Convert wavelength to optical frequency

Parameters **lambda_** : array_like
Wavelength(s) to be converted.

Returns **nu** : float or array of floats
Equivalent optical frequency.

Notes

Computes `nu = c / lambda` where c = 299792458.0, i.e., the (vacuum) speed of light in meters/second.

Examples

```
>>> from scipy.constants import lambda2nu, speed_of_light
>>> lambda2nu(np.array((1, speed_of_light)))
array([  2.99792458e+08,   1.00000000e+00])
```

`scipy.constants.`**`nu2lambda`**(*nu*)
Convert optical frequency to wavelength.

Parameters **nu** : array_like
Optical frequency to be converted.

Returns **lambda** : float or array of floats
Equivalent wavelength(s).

Notes

Computes `lambda = c / nu` where c = 299792458.0, i.e., the (vacuum) speed of light in meters/second.

Examples

```
>>> from scipy.constants import nu2lambda, speed_of_light
>>> nu2lambda(np.array((1, speed_of_light)))
array([  2.99792458e+08,   1.00000000e+00])
```

5.4.4 References

5.5 Discrete Fourier transforms (`scipy.fftpack`)

5.5.1 Fast Fourier Transforms (FFTs)

fft(x[, n, axis, overwrite_x])	Return discrete Fourier transform of real or complex sequence.
ifft(x[, n, axis, overwrite_x])	Return discrete inverse Fourier transform of real or complex sequence.
fft2(x[, shape, axes, overwrite_x])	2-D discrete Fourier transform.
ifft2(x[, shape, axes, overwrite_x])	2-D discrete inverse Fourier transform of real or complex sequence.
fftn(x[, shape, axes, overwrite_x])	Return multidimensional discrete Fourier transform.
ifftn(x[, shape, axes, overwrite_x])	Return inverse multi-dimensional discrete Fourier transform of arbitrary type sequence x.
rfft(x[, n, axis, overwrite_x])	Discrete Fourier transform of a real sequence.
irfft(x[, n, axis, overwrite_x])	Return inverse discrete Fourier transform of real sequence x.
dct(x[, type, n, axis, norm, overwrite_x])	Return the Discrete Cosine Transform of arbitrary type sequence x.

Continued on next page

Table 5.14 – continued from previous page

idct(x[, type, n, axis, norm, overwrite_x])	Return the Inverse Discrete Cosine Transform of an arbitrary type sequence.
dst(x[, type, n, axis, norm, overwrite_x])	Return the Discrete Sine Transform of arbitrary type sequence x.
idst(x[, type, n, axis, norm, overwrite_x])	Return the Inverse Discrete Sine Transform of an arbitrary type sequence.

scipy.fftpack.**fft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)` where

`y(j) = (x * exp(-2*pi*sqrt(-1)*j*np.arange(n)/n)).sum()`.

Parameters

x : array_like
: Array to Fourier transform.

n : int, optional
: Length of the Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

axis : int, optional
: Axis along which the fft's are computed; the default is over the last axis (i.e., `axis=-1`).

overwrite_x : bool, optional
: If True, the contents of *x* can be destroyed; the default is False.

Returns

z : complex ndarray
: with the elements:

```
[y(0),y(1),..,y(n/2),y(1-n/2),...,y(-1)]        if n is even
[y(0),y(1),..,y((n-1)/2),y(-(n-1)/2),...,y(-1)]  if n is odd
```

where:

```
y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k* 2*pi/n), j = 0..
↪n-1
```

Note that `y(-j) = y(n-j).conjugate()`.

See also:

ifft Inverse FFT

rfft FFT of a real sequence

Notes

The packing of the result is "standard": If `A = fft(a, n)`, then `A[0]` contains the zero-frequency term, `A[1:n/2]` contains the positive-frequency terms, and `A[n/2:]` contains the negative-frequency terms, in order of decreasingly negative frequency. So for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, -4, -3, -2, -1]. To rearrange the fft output so that the zero-frequency component is centered, like [-4, -3, -2, -1, 0, 1, 2, 3], use *fftshift*.

For *n* even, `A[n/2]` contains the sum of the positive and negative-frequency terms. For *n* even and *x* real, `A[n/2]` will always be real.

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

If the data type of *x* is real, a "real FFT" algorithm is automatically used, which roughly halves the computation time. To increase efficiency a little further, use `rfft`, which does the same calculation, but only outputs half of the symmetrical spectrum. If the data is both real and symmetrical, the `dct` can again double the efficiency, by generating half of the spectrum from half of the signal.

Examples

```
>>> from scipy.fftpack import fft, ifft
>>> x = np.arange(5)
>>> np.allclose(fft(ifft(x)), x, atol=1e-15)  # within numerical accuracy.
True
```

`scipy.fftpack.`**`ifft`**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete inverse Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)` where

`y(j) = (x * exp(2*pi*sqrt(-1)*j*np.arange(n)/n)).mean()`.

Parameters

x : array_like
: Transformed data to invert.

n : int, optional
: Length of the inverse Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

axis : int, optional
: Axis along which the ifft's are computed; the default is over the last axis (i.e., `axis=-1`).

overwrite_x : bool, optional
: If True, the contents of *x* can be destroyed; the default is False.

Returns

ifft : ndarray of floats
: The inverse discrete Fourier transform.

See also:

`fft` Forward FFT

Notes

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

If the data type of *x* is real, a "real IFFT" algorithm is automatically used, which roughly halves the computation time.

`scipy.fftpack.`**`fft2`**(*x*, *shape=None*, *axes=(-2, -1)*, *overwrite_x=False*)

2-D discrete Fourier transform.

Return the two-dimensional discrete Fourier transform of the 2-D argument *x*.

See also:

`fftn` for detailed information.

`scipy.fftpack.ifft2`(*x*, *shape=None*, *axes=(-2, -1)*, *overwrite_x=False*)
2-D discrete inverse Fourier transform of real or complex sequence.

Return inverse two-dimensional discrete Fourier transform of arbitrary type sequence x.

See `ifft` for more information.

See also:

`fft2`, `ifft`

`scipy.fftpack.fftn`(*x*, *shape=None*, *axes=None*, *overwrite_x=False*)
Return multidimensional discrete Fourier transform.

The returned array contains:

```
y[j_1,..,j_d] = sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
   x[k_1,..,k_d] * prod[i=1..d] exp(-sqrt(-1)*2*pi/n_i * j_i * k_i)
```

where d = len(x.shape) and n = x.shape. Note that `y[..., -j_i, ...] = y[..., n_i-j_i, ...].conjugate()`.

Parameters **x** : array_like
The (n-dimensional) array to transform.

shape : tuple of ints, optional
The shape of the result. If both *shape* and *axes* (see below) are None, *shape* is `x.shape`; if *shape* is None but *axes* is not None, then *shape* is `scipy.take(x.shape, axes, axis=0)`. If `shape[i] > x.shape[i]`, the i-th dimension is padded with zeros. If `shape[i] < x.shape[i]`, the i-th dimension is truncated to length `shape[i]`.

axes : array_like of ints, optional
The axes of *x* (*y* if *shape* is not None) along which the transform is applied.

overwrite_x : bool, optional
If True, the contents of *x* can be destroyed. Default is False.

Returns **y** : complex-valued n-dimensional numpy array
The (n-dimensional) DFT of the input array.

See also:

`ifftn`

Notes

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

Examples

```
>>> from scipy.fftpack import fftn, ifftn
>>> y = (-np.arange(16), 8 - np.arange(16), np.arange(16))
>>> np.allclose(y, fftn(ifftn(y)))
True
```

`scipy.fftpack.ifftn`(*x*, *shape=None*, *axes=None*, *overwrite_x=False*)
Return inverse multi-dimensional discrete Fourier transform of arbitrary type sequence x.

The returned array contains:

```
y[j_1,..,j_d] = 1/p * sum[k_1=0..n_1-1, ..., k_d=0..n_d-1]
   x[k_1,..,k_d] * prod[i=1..d] exp(sqrt(-1)*2*pi/n_i * j_i * k_i)
```

where `d = len(x.shape)`, `n = x.shape`, and `p = prod[i=1..d] n_i`.

For description of parameters see *fftn*.

See also:

fftn for detailed information.

`scipy.fftpack.`**`rfft`**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Discrete Fourier transform of a real sequence.

Parameters

x : array_like, real-valued
The data to transform.

n : int, optional
Defines the length of the Fourier transform. If *n* is not specified (the default) then `n = x.shape[axis]`. If `n < x.shape[axis]`, *x* is truncated, if `n > x.shape[axis]`, *x* is zero-padded.

axis : int, optional
The axis along which the transform is applied. The default is the last axis.

overwrite_x : bool, optional
If set to true, the contents of *x* can be overwritten. Default is False.

Returns

z : real ndarray
The returned real array contains:

```
[y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2))]              if n is
↪even
[y(0),Re(y(1)),Im(y(1)),...,Re(y(n/2)),Im(y(n/2))]   if n is odd
```

where:

```
y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k*2*pi/n)
j = 0..n-1
```

Note that `y(-j) == y(n-j).conjugate()`.

See also:

fft, *irfft*, `scipy.fftpack.basic`

Notes

Within numerical accuracy, `y == rfft(irfft(y))`.

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

Examples

```
>>> from scipy.fftpack import fft, rfft
>>> a = [9, -9, 1, 3]
>>> fft(a)
array([  4. +0.j,   8.+12.j,  16. +0.j,   8.-12.j])
>>> rfft(a)
array([  4.,   8.,  12.,  16.])
```

`scipy.fftpack.`**`irfft`**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return inverse discrete Fourier transform of real sequence x.

The contents of *x* are interpreted as the output of the *rfft* function.

Parameters
x : array_like
    Transformed data to invert.
n : int, optional
    Length of the inverse Fourier transform. If n < x.shape[axis], x is truncated. If n > x.shape[axis], x is zero-padded. The default results in n = x.shape[axis].
axis : int, optional
    Axis along which the ifft's are computed; the default is over the last axis (i.e., axis=-1).
overwrite_x : bool, optional
    If True, the contents of *x* can be destroyed; the default is False.

Returns
irfft : ndarray of floats
    The inverse discrete Fourier transform.

See also:

`rfft`, `ifft`

Notes

The returned real array contains:

```
[y(0),y(1),...,y(n-1)]
```

where for n is even:

```
y(j) = 1/n (sum[k=1..n/2-1] (x[2*k-1]+sqrt(-1)*x[2*k])
                             * exp(sqrt(-1)*j*k* 2*pi/n)
            + c.c. + x[0] + (-1)**(j) x[n-1])
```

and for n is odd:

```
y(j) = 1/n (sum[k=1..(n-1)/2] (x[2*k-1]+sqrt(-1)*x[2*k])
                             * exp(sqrt(-1)*j*k* 2*pi/n)
            + c.c. + x[0])
```

c.c. denotes complex conjugate of preceding expression.

For details on input parameters, see `rfft`.

`scipy.fftpack.`**`dct`**(*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)
Return the Discrete Cosine Transform of arbitrary type sequence x.

Parameters
x : array_like
    The input array.
type : {1, 2, 3}, optional
    Type of the DCT (see Notes). Default type is 2.
n : int, optional
    Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
axis : int, optional
    Axis along which the dct is computed; the default is over the last axis (i.e., `axis=-1`).
norm : {None, 'ortho'}, optional
    Normalization mode (see Notes). Default is None.
overwrite_x : bool, optional
    If True, the contents of *x* can be destroyed; the default is False.

Returns
y : ndarray of real
    The transformed input array.

See also:

idct	Inverse DCT

Notes

For a single dimension array `x`, `dct(x, norm='ortho')` is equal to MATLAB `dct(x)`.

There are theoretically 8 types of the DCT, only the first 3 types are implemented in scipy. 'The' DCT generally refers to DCT type 2, and 'the' Inverse DCT generally refers to DCT type 3.

Type I

There are several definitions of the DCT-I; we use the following (for `norm=None`):

```
                                          N-2
y[k] = x[0] + (-1)**k x[N-1] + 2 * sum x[n]*cos(pi*k*n/(N-1))
                                          n=1
```

Only None is supported as normalization mode for DCT-I. Note also that the DCT-I is only supported for input size > 1

Type II

There are several definitions of the DCT-II; we use the following (for `norm=None`):

```
           N-1
y[k] = 2* sum x[n]*cos(pi*k*(2n+1)/(2*N)), 0 <= k < N.
           n=0
```

If `norm='ortho'`, `y[k]` is multiplied by a scaling factor *f*:

```
f = sqrt(1/(4*N)) if k = 0,
f = sqrt(1/(2*N)) otherwise.
```

Which makes the corresponding matrix of coefficients orthonormal (`OO' = Id`).

Type III

There are several definitions, we use the following (for `norm=None`):

```
                    N-1
y[k] = x[0] + 2 * sum x[n]*cos(pi*(k+0.5)*n/N), 0 <= k < N.
                    n=1
```

or, for `norm='ortho'` and 0 <= k < N:

```
                                      N-1
y[k] = x[0] / sqrt(N) + sqrt(2/N) * sum x[n]*cos(pi*(k+0.5)*n/N)
                                      n=1
```

The (unnormalized) DCT-III is the inverse of the (unnormalized) DCT-II, up to a factor *2N*. The orthonormalized DCT-III is exactly the inverse of the orthonormalized DCT-II.

References

[R42], *[R43]*

Examples

The Type 1 DCT is equivalent to the FFT (though faster) for real, even-symmetrical inputs. The output is also real and even-symmetrical. Half of the FFT input is used to generate half of the FFT output:

```
>>> from scipy.fftpack import fft, dct
>>> fft(np.array([4., 3., 5., 10., 5., 3.])).real
array([ 30.,  -8.,   6.,  -2.,   6.,  -8.])
>>> dct(np.array([4., 3., 5., 10.]), 1)
array([ 30.,  -8.,   6.,  -2.])
```

scipy.fftpack.**idct**(*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

Return the Inverse Discrete Cosine Transform of an arbitrary type sequence.

Parameters

x : array_like
: The input array.

type : {1, 2, 3}, optional
: Type of the DCT (see Notes). Default type is 2.

n : int, optional
: Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

axis : int, optional
: Axis along which the idct is computed; the default is over the last axis (i.e., `axis=-1`).

norm : {None, 'ortho'}, optional
: Normalization mode (see Notes). Default is None.

overwrite_x : bool, optional
: If True, the contents of *x* can be destroyed; the default is False.

Returns

idct : ndarray of real
: The transformed input array.

See also:

dct Forward DCT

Notes

For a single dimension array *x*, `idct(x, norm='ortho')` is equal to MATLAB `idct(x)`.

'The' IDCT is the IDCT of type 2, which is the same as DCT of type 3.

IDCT of type 1 is the DCT of type 1, IDCT of type 2 is the DCT of type 3, and IDCT of type 3 is the DCT of type 2. For the definition of these types, see *dct*.

Examples

The Type 1 DCT is equivalent to the DFT for real, even-symmetrical inputs. The output is also real and even-symmetrical. Half of the IFFT input is used to generate half of the IFFT output:

```
>>> from scipy.fftpack import ifft, idct
>>> ifft(np.array([ 30.,  -8.,   6.,  -2.,   6.,  -8.])).real
array([  4.,   3.,   5.,  10.,   5.,   3.])
>>> idct(np.array([ 30.,  -8.,   6.,  -2.]), 1) / 6
array([  4.,   3.,   5.,  10.])
```

scipy.fftpack.**dst**(*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

Return the Discrete Sine Transform of arbitrary type sequence x.

Parameters

x : array_like
: The input array.

type : {1, 2, 3}, optional
: Type of the DST (see Notes). Default type is 2.

n : int, optional

Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

axis : int, optional
Axis along which the dst is computed; the default is over the last axis (i.e., `axis=-1`).

norm : {None, 'ortho'}, optional
Normalization mode (see Notes). Default is None.

overwrite_x : bool, optional
If True, the contents of *x* can be destroyed; the default is False.

Returns **dst** : ndarray of reals
The transformed input array.

See also:

`idst` Inverse DST

Notes

For a single dimension array `x`.

There are theoretically 8 types of the DST for different combinations of even/odd boundary conditions and boundary off sets *[R44]*, only the first 3 types are implemented in scipy.

Type I

There are several definitions of the DST-I; we use the following for `norm=None`. DST-I assumes the input is odd around n=-1 and n=N.

```
           N-1
y[k] = 2 * sum x[n]*sin(pi*(k+1)*(n+1)/(N+1))
           n=0
```

Only None is supported as normalization mode for DCT-I. Note also that the DCT-I is only supported for input size > 1 The (unnormalized) DCT-I is its own inverse, up to a factor *2(N+1)*.

Type II

There are several definitions of the DST-II; we use the following for `norm=None`. DST-II assumes the input is odd around n=-1/2 and n=N-1/2; the output is odd around k=-1 and even around k=N-1

```
          N-1
y[k] = 2* sum x[n]*sin(pi*(k+1)*(n+0.5)/N), 0 <= k < N.
          n=0
```

if `norm='ortho'`, `y[k]` is multiplied by a scaling factor *f*

```
f = sqrt(1/(4*N)) if k == 0
f = sqrt(1/(2*N)) otherwise.
```

Type III

There are several definitions of the DST-III, we use the following (for `norm=None`). DST-III assumes the input is odd around n=-1 and even around n=N-1

```
                            N-2
y[k] = x[N-1]*(-1)**k + 2* sum x[n]*sin(pi*(k+0.5)*(n+1)/N), 0 <= k < N.
                            n=0
```

The (unnormalized) DCT-III is the inverse of the (unnormalized) DCT-II, up to a factor *2N*. The orthonormalized DST-III is exactly the inverse of the orthonormalized DST-II.

New in version 0.11.0.

References

[R44]

`scipy.fftpack.idst` (*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

Return the Inverse Discrete Sine Transform of an arbitrary type sequence.

Parameters

x : array_like
: The input array.

type : {1, 2, 3}, optional
: Type of the DST (see Notes). Default type is 2.

n : int, optional
: Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.

axis : int, optional
: Axis along which the idst is computed; the default is over the last axis (i.e., `axis=-1`).

norm : {None, 'ortho'}, optional
: Normalization mode (see Notes). Default is None.

overwrite_x : bool, optional
: If True, the contents of *x* can be destroyed; the default is False.

Returns

idst : ndarray of real
: The transformed input array.

See also:

`dst` Forward DST

Notes

'The' IDST is the IDST of type 2, which is the same as DST of type 3.

IDST of type 1 is the DST of type 1, IDST of type 2 is the DST of type 3, and IDST of type 3 is the DST of type 2. For the definition of these types, see `dst`.

New in version 0.11.0.

5.5.2 Differential and pseudo-differential operators

diff(x[, order, period, _cache])	Return k-th derivative (or integral) of a periodic sequence x.
tilbert(x, h[, period, _cache])	Return h-Tilbert transform of a periodic sequence x.
itilbert(x, h[, period, _cache])	Return inverse h-Tilbert transform of a periodic sequence x.
hilbert(x[, _cache])	Return Hilbert transform of a periodic sequence x.
ihilbert(x)	Return inverse Hilbert transform of a periodic sequence x.
cs_diff(x, a, b[, period, _cache])	Return (a,b)-cosh/sinh pseudo-derivative of a periodic sequence.
sc_diff(x, a, b[, period, _cache])	Return (a,b)-sinh/cosh pseudo-derivative of a periodic sequence x.
ss_diff(x, a, b[, period, _cache])	Return (a,b)-sinh/sinh pseudo-derivative of a periodic sequence x.
cc_diff(x, a, b[, period, _cache])	Return (a,b)-cosh/cosh pseudo-derivative of a periodic sequence.

Continued on next page

Table 5.15 – continued from previous page

shift(x, a[, period, _cache])	Shift periodic sequence x by a: y(u) = x(u+a).

scipy.fftpack.**diff**(*x*, *order=1*, *period=None*, *_cache={}*)

Return k-th derivative (or integral) of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = pow(sqrt(-1)*j*2*pi/period, order) * x_j
y_0 = 0 if order is not 0.
```

Parameters

x : array_like
: Input array.

order : int, optional
: The order of differentiation. Default order is 1. If order is negative, then integration is carried out under the assumption that `x_0 == 0`.

period : float, optional
: The assumed period of the sequence. Default is `2*pi`.

Notes

If `sum(x, axis=0) = 0` then `diff(diff(x, k), -k) == x` (within numerical accuracy).

For odd order and even `len(x)`, the Nyquist mode is taken zero.

scipy.fftpack.**tilbert**(*x*, *h*, *period=None*, *_cache={}*)

Return h-Tilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = sqrt(-1)*coth(j*h*2*pi/period) * x_j
y_0 = 0
```

Parameters

x : array_like
: The input array to transform.

h : float
: Defines the parameter of the Tilbert transform.

period : float, optional
: The assumed period of the sequence. Default period is `2*pi`.

Returns

tilbert : ndarray
: The result of the transform.

Notes

If `sum(x, axis=0) == 0` and `n = len(x)` is odd then `tilbert(itilbert(x)) == x`.

If `2 * pi * h / period` is approximately 10 or larger, then numerically `tilbert == hilbert` (theoretically oo-Tilbert == Hilbert).

For even `len(x)`, the Nyquist mode of `x` is taken zero.

scipy.fftpack.**itilbert**(*x*, *h*, *period=None*, *_cache={}*)

Return inverse h-Tilbert transform of a periodic sequence x.

If `x_j` and `y_j` are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = -sqrt(-1)*tanh(j*h*2*pi/period) * x_j
y_0 = 0
```

For more details, see `tilbert`.

`scipy.fftpack.hilbert(x, _cache={})`
Return Hilbert transform of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = sqrt(-1)*sign(j) * x_j
y_0 = 0
```

Parameters **x** : array_like
The input array, should be periodic.
_cache : dict, optional
Dictionary that contains the kernel used to do a convolution with.

Returns **y** : ndarray
The transformed input.

See also:

`scipy.signal.hilbert`
Compute the analytic signal, using the Hilbert transform.

Notes

If `sum(x, axis=0) == 0` then `hilbert(ihilbert(x)) == x`.

For even len(x), the Nyquist mode of x is taken zero.

The sign of the returned transform does not have a factor -1 that is more often than not found in the definition of the Hilbert transform. Note also that `scipy.signal.hilbert` does have an extra -1 factor compared to this function.

`scipy.fftpack.ihilbert(x)`
Return inverse Hilbert transform of a periodic sequence x.

If `x_j` and `y_j` are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = -sqrt(-1)*sign(j) * x_j
y_0 = 0
```

`scipy.fftpack.cs_diff(x, a, b, period=None, _cache={})`
Return (a,b)-cosh/sinh pseudo-derivative of a periodic sequence.

If `x_j` and `y_j` are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = -sqrt(-1)*cosh(j*a*2*pi/period)/sinh(j*b*2*pi/period) * x_j
y_0 = 0
```

Parameters **x** : array_like
The array to take the pseudo-derivative from.
a, b : float
Defines the parameters of the cosh/sinh pseudo-differential operator.
period : float, optional
The period of the sequence. Default period is `2*pi`.

Returns **cs_diff** : ndarray
Pseudo-derivative of periodic sequence *x*.

Notes

For even len(*x*), the Nyquist mode of *x* is taken as zero.

scipy.fftpack.**sc_diff**(*x*, *a*, *b*, *period=None*, *_cache={}*)

Return (a,b)-sinh/cosh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = sqrt(-1)*sinh(j*a*2*pi/period)/cosh(j*b*2*pi/period) * x_j
y_0 = 0
```

Parameters **x** : array_like
Input array.

a,b : float
Defines the parameters of the sinh/cosh pseudo-differential operator.

period : float, optional
The period of the sequence x. Default is 2*pi.

Notes

`sc_diff(cs_diff(x,a,b),b,a) == x` For even `len(x)`, the Nyquist mode of x is taken as zero.

scipy.fftpack.**ss_diff**(*x*, *a*, *b*, *period=None*, *_cache={}*)

Return (a,b)-sinh/sinh pseudo-derivative of a periodic sequence x.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = sinh(j*a*2*pi/period)/sinh(j*b*2*pi/period) * x_j
y_0 = a/b * x_0
```

Parameters **x** : array_like
The array to take the pseudo-derivative from.

a,b
Defines the parameters of the sinh/sinh pseudo-differential operator.

period : float, optional
The period of the sequence x. Default is `2*pi`.

Notes

`ss_diff(ss_diff(x,a,b),b,a) == x`

scipy.fftpack.**cc_diff**(*x*, *a*, *b*, *period=None*, *_cache={}*)

Return (a,b)-cosh/cosh pseudo-derivative of a periodic sequence.

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = cosh(j*a*2*pi/period)/cosh(j*b*2*pi/period) * x_j
```

Parameters **x** : array_like
The array to take the pseudo-derivative from.

a,b : float
Defines the parameters of the sinh/sinh pseudo-differential operator.

period : float, optional
The period of the sequence x. Default is `2*pi`.

Returns **cc_diff** : ndarray
Pseudo-derivative of periodic sequence *x*.

Notes

`cc_diff(cc_diff(x,a,b),b,a) == x`

`scipy.fftpack.shift(x, a, period=None, _cache={})`

Shift periodic sequence x by a: y(u) = x(u+a).

If x_j and y_j are Fourier coefficients of periodic functions x and y, respectively, then:

```
y_j = exp(j*a*2*pi/period*sqrt(-1)) * x_f
```

Parameters

x : array_like
: The array to take the pseudo-derivative from.

a : float
: Defines the parameters of the sinh/sinh pseudo-differential

period : float, optional
: The period of the sequences x and y. Default period is `2*pi`.

5.5.3 Helper functions

fftshift(x[, axes])	Shift the zero-frequency component to the center of the spectrum.
ifftshift(x[, axes])	The inverse of *fftshift*.
fftfreq(n[, d])	Return the Discrete Fourier Transform sample frequencies.
rfftfreq(n[, d])	DFT sample frequencies (for usage with rfft, irfft).
next_fast_len(target)	Find the next fast size of input data to *fft*, for zero-padding, etc.

`scipy.fftpack.fftshift(x, axes=None)`

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

Parameters

x : array_like
: Input array.

axes : int or shape tuple, optional
: Axes over which to shift. Default is None, which shifts all axes.

Returns

y : ndarray
: The shifted array.

See also:

ifftshift The inverse of *fftshift*.

Examples

```
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2.,  3.,  4., -5., -4., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2.,  0.,  1.],
       [-4.,  3.,  4.],
       [-1., -3., -2.]])
```

scipy.fftpack.**ifftshift**(*x*, *axes=None*)

The inverse of *fftshift*. Although identical for even-length *x*, the functions differ by one sample for odd-length *x*.

Parameters **x** : array_like
Input array.
axes : int or shape tuple, optional
Axes over which to calculate. Defaults to None, which shifts all axes.

Returns **y** : ndarray
The shifted array.

See also:

fftshift Shift zero-frequency component to the center of the spectrum.

Examples

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
```

scipy.fftpack.**fftfreq**(*n*, *d=1.0*)

Return the Discrete Fourier Transform sample frequencies.

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
```

Parameters **n** : int
Window length.
d : scalar, optional
Sample spacing (inverse of the sampling rate). Defaults to 1.

Returns **f** : ndarray
Array of length *n* containing the sample frequencies.

Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

`scipy.fftpack.`**`rfftfreq`**`(`*n*, *d=1.0*`)`

DFT sample frequencies (for usage with rfft, irfft).

The returned float array contains the frequency bins in cycles/unit (with zero at the start) given a window length *n* and a sample spacing *d*:

```
f = [0,1,1,2,2,...,n/2-1,n/2-1,n/2]/(d*n)   if n is even
f = [0,1,1,2,2,...,n/2-1,n/2-1,n/2,n/2]/(d*n)   if n is odd
```

Parameters **n** : int

Window length.

d : scalar, optional

Sample spacing. Default is 1.

Returns **out** : ndarray

The array of length *n*, containing the sample frequencies.

Examples

```
>>> from scipy import fftpack
>>> sig = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> sig_fft = fftpack.rfft(sig)
>>> n = sig_fft.size
>>> timestep = 0.1
>>> freq = fftpack.rfftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  1.25,  2.5 ,  2.5 ,  3.75,  3.75,  5.  ])
```

`scipy.fftpack.`**`next_fast_len`**`(`*target*`)`

Find the next fast size of input data to `fft`, for zero-padding, etc.

SciPy's FFTPACK has efficient functions for radix {2, 3, 4, 5}, so this returns the next composite of the prime factors 2, 3, and 5 which is greater than or equal to *target*. (These are also known as 5-smooth numbers, regular numbers, or Hamming numbers.)

Parameters **target** : int

Length to start searching from. Must be a positive integer.

Returns **out** : int

The first 5-smooth number greater than or equal to *target*.

Notes

New in version 0.18.0.

Examples

On a particular machine, an FFT of prime length takes 133 ms:

```
>>> from scipy import fftpack
>>> min_len = 10007  # prime length is worst case for speed
>>> a = np.random.randn(min_len)
>>> b = fftpack.fft(a)
```

Zero-padding to the next 5-smooth length reduces computation time to 211 us, a speedup of 630 times:

```
>>> fftpack.helper.next_fast_len(min_len)
10125
>>> b = fftpack.fft(a, 10125)
```

Rounding up to the next power of 2 is not optimal, taking 367 us to compute, 1.7 times as long as the 5-smooth size:

```
>>> b = fftpack.fft(a, 16384)
```

Note that `fftshift`, `ifftshift` and `fftfreq` are numpy functions exposed by `fftpack`; importing them from `numpy` should be preferred.

5.5.4 Convolutions (`scipy.fftpack.convolve`)

convolve(x,omega,[swap_real_imag,overwrite_x])	Wrapper for `convolve`.
convolve_z(x,omega_real,omega_imag,[overwrite_x])	Wrapper for `convolve_z`.
init_convolution_kernel(...)	Wrapper for `init_convolution_kernel`.
destroy_convolve_cache()	Wrapper for `destroy_convolve_cache`.

`scipy.fftpack.convolve.`**`convolve`**(*x*, *omega*[, *swap_real_imag*, *overwrite_x*]) **= <fortran object>**

Wrapper for `convolve`.

Parameters **x** : input rank-1 array('d') with bounds (n)
omega : input rank-1 array('d') with bounds (n)

Returns **y** : rank-1 array('d') with bounds (n) and x storage

Other Parameters

overwrite_x : input int, optional
Default: 0

swap_real_imag : input int, optional
Default: 0

`scipy.fftpack.convolve.`**`convolve_z`**(*x*, *omega_real*, *omega_imag*[, *overwrite_x*]) **= <fortran object>**

Wrapper for `convolve_z`.

Parameters **x** : input rank-1 array('d') with bounds (n)
omega_real : input rank-1 array('d') with bounds (n)
omega_imag : input rank-1 array('d') with bounds (n)

Returns **y** : rank-1 array('d') with bounds (n) and x storage

Other Parameters

overwrite_x : input int, optional
Default: 0

`scipy.fftpack.convolve.`**`init_convolution_kernel`**(*n*, *kernel_func*[, *d*, *zero_nyquist*, *kernel_func_extra_args*]) **= <fortran object>**

Wrapper for `init_convolution_kernel`.

Parameters **n** : input int
kernel_func : call-back function

Returns **omega** : rank-1 array('d') with bounds (n)

Other Parameters

d : input int, optional
Default: 0

kernel_func_extra_args : input tuple, optional
Default: ()

zero_nyquist : input int, optional
Default: d%2

Notes

Call-back functions:

```
def kernel_func(k): return kernel_func
Required arguments:
  k : input int
Return objects:
  kernel_func : float
```

scipy.fftpack.convolve.**destroy_convolve_cache** = <fortran object>
Wrapper for `destroy_convolve_cache`.

5.6 Integration and ODEs (`scipy.integrate`)

5.6.1 Integrating functions, given function object

quad(func, a, b[, args, full_output, ...])	Compute a definite integral.
dblquad(func, a, b, gfun, hfun[, args, ...])	Compute a double integral.
tplquad(func, a, b, gfun, hfun, qfun, rfun)	Compute a triple (definite) integral.
nquad(func, ranges[, args, opts, full_output])	Integration over multiple variables.
fixed_quad(func, a, b[, args, n])	Compute a definite integral using fixed-order Gaussian quadrature.
quadrature(func, a, b[, args, tol, rtol, ...])	Compute a definite integral using fixed-tolerance Gaussian quadrature.
romberg(function, a, b[, args, tol, rtol, ...])	Romberg integration of a callable function or method.
quad_explain([output])	Print extra information about integrate.quad() parameters and returns.
newton_cotes(rn[, equal])	Return weights and error coefficient for Newton-Cotes integration.
IntegrationWarning	Warning on issues during integration.

scipy.integrate.**quad**(*func*, *a*, *b*, *args=()*, *full_output=0*, *epsabs=1.49e-08*, *epsrel=1.49e-08*, *limit=50*, *points=None*, *weight=None*, *wvar=None*, *wopts=None*, *maxp1=50*, *limlst=50*)
Compute a definite integral.

Integrate func from *a* to *b* (possibly infinite interval) using a technique from the Fortran library QUADPACK.

Parameters **func** : {function, scipy.LowLevelCallable}
A Python function or method to integrate. If *func* takes many arguments, it is integrated along the axis corresponding to the first argument.

If the user desires improved integration performance, then f may be a *scipy.LowLevelCallable* with one of the signatures:

```
double func(double x)
double func(double x, void *user_data)
double func(int n, double *xx)
double func(int n, double *xx, void *user_data)
```

The `user_data` is the data contained in the *scipy.LowLevelCallable*. In the call forms with `xx`, `n` is the length of the `xx` array which contains `xx[0] == x` and the rest of the items are numbers contained in the `args` argument of quad.
In addition, certain ctypes call signatures are supported for backward compatibility, but those should not be used in new code.

a : float
Lower limit of integration (use -numpy.inf for -infinity).

b : float
Upper limit of integration (use numpy.inf for +infinity).

args : tuple, optional
Extra arguments to pass to *func*.

full_output : int, optional
Non-zero to return a dictionary of integration information. If non-zero, warning messages are also suppressed and the message is appended to the output tuple.

Returns

y : float
The integral of func from *a* to *b*.

abserr : float
An estimate of the absolute error in the result.

infodict : dict
A dictionary containing additional information. Run scipy.integrate.quad_explain() for more information.

message
A convergence message.

explain
Appended only with 'cos' or 'sin' weighting and infinite integration limits, it contains an explanation of the codes in infodict['ierlst']

Other Parameters

epsabs : float or int, optional
Absolute error tolerance.

epsrel : float or int, optional
Relative error tolerance.

limit : float or int, optional
An upper bound on the number of subintervals used in the adaptive algorithm.

points : (sequence of floats,ints), optional
A sequence of break points in the bounded integration interval where local difficulties of the integrand may occur (e.g., singularities, discontinuities). The sequence does not have to be sorted.

weight : float or int, optional
String indicating weighting function. Full explanation for this and the remaining arguments can be found below.

wvar : optional
Variables for use with weighting functions.

wopts : optional
Optional input for reusing Chebyshev moments.

maxp1 : float or int, optional
An upper bound on the number of Chebyshev moments.

limlst : int, optional

Upper bound on the number of cycles (>=3) for use with a sinusoidal weighting and an infinite end-point.

See also:

dblquad double integral

tplquad triple integral

nquad n-dimensional integrals (uses `quad` recursively)

fixed_quad fixed-order Gaussian quadrature

quadrature adaptive Gaussian quadrature

odeint ODE integrator

ode ODE integrator

simps integrator for sampled data

romb integrator for sampled data

scipy.special
for coefficients and roots of orthogonal polynomials

Notes

Extra information for quad() inputs and outputs

If full_output is non-zero, then the third output argument (infodict) is a dictionary with entries as tabulated below. For infinite limits, the range is transformed to (0,1) and the optional outputs are given with respect to this transformed range. Let M be the input argument limit and let K be infodict['last']. The entries are:

'neval' The number of function evaluations.

'last' The number, K, of subintervals produced in the subdivision process.

'alist' A rank-1 array of length M, the first K elements of which are the left end points of the subintervals in the partition of the integration range.

'blist' A rank-1 array of length M, the first K elements of which are the right end points of the subintervals.

'rlist' A rank-1 array of length M, the first K elements of which are the integral approximations on the subintervals.

'elist' A rank-1 array of length M, the first K elements of which are the moduli of the absolute error estimates on the subintervals.

'iord' A rank-1 integer array of length M, the first L elements of which are pointers to the error estimates over the subintervals with `L=K` if `K<=M/2+2` or `L=M+1-K` otherwise. Let I be the sequence `infodict['iord']` and let E be the sequence `infodict['elist']`. Then `E[I[1]], ..., E[I[L]]` forms a decreasing sequence.

If the input argument points is provided (i.e. it is not None), the following additional outputs are placed in the output dictionary. Assume the points sequence is of length P.

'pts' A rank-1 array of length P+2 containing the integration limits and the break points of the intervals in ascending order. This is an array giving the subintervals over which integration will occur.

'level' A rank-1 integer array of length M (=limit), containing the subdivision levels of the subintervals, i.e., if (aa,bb) is a subinterval of `(pts[1], pts[2])` where `pts[0]` and `pts[2]` are adjacent elements of `infodict['pts']`, then (aa,bb) has level l if `|bb-aa| = |pts[2]-pts[1]| * 2**(-l)`.

'ndin' A rank-1 integer array of length P+2. After the first integration over the intervals (pts[1], pts[2]), the error estimates over some of the intervals may have been increased artificially in order to put their subdivision forward. This array has ones in slots corresponding to the subintervals for which this happens.

Weighting the integrand

The input variables, *weight* and *wvar*, are used to weight the integrand by a select list of functions. Different integration methods are used to compute the integral with these weighting functions. The possible values of weight and the corresponding weighting functions are.

`weight`	Weight function used	`wvar`
'cos'	cos(w*x)	wvar = w
'sin'	sin(w*x)	wvar = w
'alg'	g(x) = ((x-a)**alpha)*((b-x)**beta)	wvar = (alpha, beta)
'alg-loga'	g(x)*log(x-a)	wvar = (alpha, beta)
'alg-logb'	g(x)*log(b-x)	wvar = (alpha, beta)
'alg-log'	g(x)*log(x-a)*log(b-x)	wvar = (alpha, beta)
'cauchy'	1/(x-c)	wvar = c

wvar holds the parameter w, (alpha, beta), or c depending on the weight selected. In these expressions, a and b are the integration limits.

For the 'cos' and 'sin' weighting, additional inputs and outputs are available.

For finite integration limits, the integration is performed using a Clenshaw-Curtis method which uses Chebyshev moments. For repeated calculations, these moments are saved in the output dictionary:

'momcom' The maximum level of Chebyshev moments that have been computed, i.e., if `M_c` is `infodict['momcom']` then the moments have been computed for intervals of length `|b-a| * 2**(-l), l=0,1,...,M_c`.

'nnlog' A rank-1 integer array of length M(=limit), containing the subdivision levels of the subintervals, i.e., an element of this array is equal to l if the corresponding subinterval is `|b-a|* 2**(-l)`.

'chebmo' A rank-2 array of shape (25, maxp1) containing the computed Chebyshev moments. These can be passed on to an integration over the same interval by passing this array as the second element of the sequence wopts and passing infodict['momcom'] as the first element.

If one of the integration limits is infinite, then a Fourier integral is computed (assuming w neq 0). If full_output is 1 and a numerical error is encountered, besides the error message attached to the output tuple, a dictionary is also appended to the output tuple which translates the error codes in the array `info['ierlst']` to English messages. The output information dictionary contains the following entries instead of 'last', 'alist', 'blist', 'rlist', and 'elist':

'lst' The number of subintervals needed for the integration (call it `K_f`).

'rslst' A rank-1 array of length M_f=limlst, whose first `K_f` elements contain the integral contribution over the interval `(a+(k-1)c, a+kc)` where `c = (2*floor(|w|) + 1) * pi / |w|` and `k=1,2,...,K_f`.

'erlst' A rank-1 array of length `M_f` containing the error estimate corresponding to the interval in the same position in `infodict['rslist']`.

'ierlst' A rank-1 integer array of length `M_f` containing an error flag corresponding to the interval in the same position in `infodict['rslist']`. See the explanation dictionary (last entry in the output tuple) for the meaning of the codes.

Examples

Calculate $\int_0^4 x^2 dx$ and compare with an analytic result

```
>>> from scipy import integrate
>>> x2 = lambda x: x**2
>>> integrate.quad(x2, 0, 4)
(21.333333333333332, 2.3684757858670003e-13)
>>> print(4**3 / 3.)  # analytical result
21.3333333333
```

Calculate $\int_0^\infty e^{-x} dx$

```
>>> invexp = lambda x: np.exp(-x)
>>> integrate.quad(invexp, 0, np.inf)
(1.0, 5.842605999138044e-11)
```

```
>>> f = lambda x,a : a*x
>>> y, err = integrate.quad(f, 0, 1, args=(1,))
>>> y
0.5
>>> y, err = integrate.quad(f, 0, 1, args=(3,))
>>> y
1.5
```

Calculate $\int_0^1 x^2 + y^2 dx$ with ctypes, holding y parameter as 1:

```
testlib.c =>
    double func(int n, double args[n]){
        return args[0]*args[0] + args[1]*args[1];}
compile to library testlib.*
```

```
from scipy import integrate
import ctypes
lib = ctypes.CDLL('/home/.../testlib.*') #use absolute path
lib.func.restype = ctypes.c_double
lib.func.argtypes = (ctypes.c_int,ctypes.c_double)
integrate.quad(lib.func,0,1,(1))
#(1.3333333333333333, 1.4802973661668752e-14)
print((1.0**3/3.0 + 1.0) - (0.0**3/3.0 + 0.0)) #Analytic result
# 1.3333333333333333
```

`scipy.integrate.dblquad`(*func*, *a*, *b*, *gfun*, *hfun*, *args=()*, *epsabs=1.49e-08*, *epsrel=1.49e-08*)

Compute a double integral.

Return the double (definite) integral of `func(y, x)` from `x = a..b` and `y = gfun(x)..hfun(x)`.

Parameters

func : callable
: A Python function or method of at least two variables: y must be the first argument and x the second argument.

a, b : float
: The limits of integration in x: $a < b$

gfun : callable
: The lower boundary curve in y which is a function taking a single floating point argument (x) and returning a floating point result: a lambda function can be useful here.

hfun : callable
: The upper boundary curve in y (same requirements as *gfun*).

args : sequence, optional
: Extra arguments to pass to *func*.

epsabs : float, optional

Absolute tolerance passed directly to the inner 1-D quadrature integration. Default is 1.49e-8.

epsrel : float, optional
Relative tolerance of the inner 1-D integrals. Default is 1.49e-8.

Returns **y** : float
The resultant integral.

abserr : float
An estimate of the error.

See also:

quad single integral

tplquad triple integral

nquad N-dimensional integrals

fixed_quad fixed-order Gaussian quadrature

quadrature adaptive Gaussian quadrature

odeint ODE integrator

ode ODE integrator

simps integrator for sampled data

romb integrator for sampled data

scipy.special
for coefficients and roots of orthogonal polynomials

`scipy.integrate.`**`tplquad`**(*func*, *a*, *b*, *gfun*, *hfun*, *qfun*, *rfun*, *args=()*, *epsabs=1.49e-08*, *epsrel=1.49e-08*)

Compute a triple (definite) integral.

Return the triple integral of `func(z, y, x)` from `x = a..b`, `y = gfun(x)..hfun(x)`, and `z = qfun(x,y)..rfun(x,y)`.

Parameters **func** : function
A Python function or method of at least three variables in the order (z, y, x).

a, b : float
The limits of integration in x: $a < b$

gfun : function
The lower boundary curve in y which is a function taking a single floating point argument (x) and returning a floating point result: a lambda function can be useful here.

hfun : function
The upper boundary curve in y (same requirements as *gfun*).

qfun : function
The lower boundary surface in z. It must be a function that takes two floats in the order (x, y) and returns a float.

rfun : function
The upper boundary surface in z. (Same requirements as *qfun*.)

args : tuple, optional
Extra arguments to pass to *func*.

epsabs : float, optional
Absolute tolerance passed directly to the innermost 1-D quadrature integration. Default is 1.49e-8.

epsrel : float, optional
Relative tolerance of the innermost 1-D integrals. Default is 1.49e-8.

Returns **y** : float
The resultant integral.
abserr : float
An estimate of the error.

See also:

quad Adaptive quadrature using QUADPACK

quadrature Adaptive Gaussian quadrature

fixed_quad Fixed-order Gaussian quadrature

dblquad Double integrals

nquad N-dimensional integrals

romb Integrators for sampled data

simps Integrators for sampled data

ode ODE integrators

odeint ODE integrators

scipy.special
For coefficients and roots of orthogonal polynomials

scipy.integrate.**nquad**(*func*, *ranges*, *args=None*, *opts=None*, *full_output=False*)
Integration over multiple variables.

Wraps *quad* to enable integration over multiple variables. Various options allow improved integration of discontinuous functions, as well as the use of weighted integration, and generally finer control of the integration process.

Parameters **func** : {callable, scipy.LowLevelCallable}
The function to be integrated. Has arguments of `x0, ... xn`, `t0, tm`, where integration is carried out over `x0, ... xn`, which must be floats. Function signature should be `func(x0, x1, ..., xn, t0, t1, ..., tm)`. Integration is carried out in order. That is, integration over `x0` is the innermost integral, and `xn` is the outermost.
If the user desires improved integration performance, then *f* may be a *scipy.LowLevelCallable* with one of the signatures:

```
double func(int n, double *xx)
double func(int n, double *xx, void *user_data)
```

where `n` is the number of extra parameters and args is an array of doubles of the additional parameters, the `xx` array contains the coordinates. The `user_data` is the data contained in the *scipy.LowLevelCallable*.
ranges : iterable object
Each element of ranges may be either a sequence of 2 numbers, or else a callable that returns such a sequence. `ranges[0]` corresponds to integration over x0, and so on. If an element of ranges is a callable, then it will be called with all of the integration arguments available, as well as any parametric arguments. e.g. if `func = f(x0, x1, x2, t0, t1)`, then `ranges[0]` may be defined as either `(a, b)` or else as `(a, b) = range0(x1, x2, t0, t1)`.
args : iterable object, optional
Additional arguments `t0, ..., tn`, required by *func*, *ranges*, and `opts`.
opts : iterable object or dict, optional

Options to be passed to *quad*. May be empty, a dict, or a sequence of dicts or functions that return a dict. If empty, the default options from scipy.integrate.quad are used. If a dict, the same options are used for all levels of integraion. If a sequence, then each element of the sequence corresponds to a particular integration. e.g. opts[0] corresponds to integration over x0, and so on. If a callable, the signature must be the same as for `ranges`. The available options together with their default values are:

- epsabs = 1.49e-08
- epsrel = 1.49e-08
- limit = 50
- points = None
- weight = None
- wvar = None
- wopts = None

For more information on these options, see *quad* and *quad_explain*.

full_output : bool, optional

Partial implementation of `full_output` from scipy.integrate.quad. The number of integrand function evaluations `neval` can be obtained by setting `full_output=True` when calling nquad.

Returns

result : float

The result of the integration.

abserr : float

The maximum of the estimates of the absolute error in the various integration results.

out_dict : dict, optional

A dict containing additional information on the integration.

See also:

quad 1-dimensional numerical integration

dblquad, *tplquad*

fixed_quad fixed-order Gaussian quadrature

quadrature adaptive Gaussian quadrature

Examples

```
>>> from scipy import integrate
>>> func = lambda x0,x1,x2,x3 : x0**2 + x1*x2 - x3**3 + np.sin(x0) + (
...                         1 if (x0-.2*x3-.5-.25*x1>0) else 0)
>>> points = [[lambda x1,x2,x3 : 0.2*x3 + 0.5 + 0.25*x1], [], [], []]
>>> def opts0(*args, **kwargs):
...     return {'points':[0.2*args[2] + 0.5 + 0.25*args[0]]}
>>> integrate.nquad(func, [[0,1], [-1,1], [.13,.8], [-.15,1]],
...                 opts=[opts0,{},{},{}], full_output=True)
(1.5267454070738633, 2.9437360001402324e-14, {'neval': 388962})
```

```
>>> scale = .1
>>> def func2(x0, x1, x2, x3, t0, t1):
...     return x0*x1*x3**2 + np.sin(x2) + 1 + (1 if x0+t1*x1-t0>0 else 0)
>>> def lim0(x1, x2, x3, t0, t1):
...     return [scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) - 1,
...             scale * (x1**2 + x2 + np.cos(x3)*t0*t1 + 1) + 1]
>>> def lim1(x2, x3, t0, t1):
...     return [scale * (t0*x2 + t1*x3) - 1,
...             scale * (t0*x2 + t1*x3) + 1]
>>> def lim2(x3, t0, t1):
```

```
...     return [scale * (x3 + t0**2*t1**3) - 1,
...             scale * (x3 + t0**2*t1**3) + 1]
>>> def lim3(t0, t1):
...     return [scale * (t0+t1) - 1, scale * (t0+t1) + 1]
>>> def opts0(x1, x2, x3, t0, t1):
...     return {'points' : [t0 - t1*x1]}
>>> def opts1(x2, x3, t0, t1):
...     return {}
>>> def opts2(x3, t0, t1):
...     return {}
>>> def opts3(t0, t1):
...     return {}
>>> integrate.nquad(func2, [lim0, lim1, lim2, lim3], args=(0,0),
...                 opts=[opts0, opts1, opts2, opts3])
(25.066666666666666, 2.7829590483937256e-13)
```

`scipy.integrate.`**`fixed_quad`**(*func*, *a*, *b*, *args=()*, *n=5*)

Compute a definite integral using fixed-order Gaussian quadrature.

Integrate *func* from *a* to *b* using Gaussian quadrature of order *n*.

Parameters

func : callable
: A Python function or method to integrate (must accept vector inputs). If integrating a vector-valued function, the returned array must have shape `(..., len(x))`.

a : float
: Lower limit of integration.

b : float
: Upper limit of integration.

args : tuple, optional
: Extra arguments to pass to function, if any.

n : int, optional
: Order of quadrature integration. Default is 5.

Returns

val : float
: Gaussian quadrature approximation to the integral

none : None
: Statically returned value of None

See also:

`quad`	adaptive quadrature using QUADPACK
`dblquad`	double integrals
`tplquad`	triple integrals
`romberg`	adaptive Romberg quadrature
`quadrature`	adaptive Gaussian quadrature
`romb`	integrators for sampled data
`simps`	integrators for sampled data
`cumtrapz`	cumulative integration for sampled data
`ode`	ODE integrator
`odeint`	ODE integrator

`scipy.integrate.quadrature`(*func*, *a*, *b*, *args=()*, *tol=1.49e-08*, *rtol=1.49e-08*, *maxiter=50*, *vec_func=True*, *miniter=1*)

Compute a definite integral using fixed-tolerance Gaussian quadrature.

Integrate *func* from *a* to *b* using Gaussian quadrature with absolute tolerance *tol*.

Parameters

func : function
: A Python function or method to integrate.

a : float
: Lower limit of integration.

b : float
: Upper limit of integration.

args : tuple, optional
: Extra arguments to pass to function.

tol, rtol : float, optional
: Iteration stops when error between last two iterates is less than *tol* OR the relative change is less than *rtol*.

maxiter : int, optional
: Maximum order of Gaussian quadrature.

vec_func : bool, optional
: True or False if func handles arrays as arguments (is a "vector" function). Default is True.

miniter : int, optional
: Minimum order of Gaussian quadrature.

Returns

val : float
: Gaussian quadrature approximation (within tolerance) to integral.

err : float
: Difference between last two estimates of the integral.

See also:

`romberg` adaptive Romberg quadrature

`fixed_quad` fixed-order Gaussian quadrature

`quad` adaptive quadrature using QUADPACK

`dblquad` double integrals

`tplquad` triple integrals

`romb` integrator for sampled data

`simps` integrator for sampled data

`cumtrapz` cumulative integration for sampled data

`ode` ODE integrator

`odeint` ODE integrator

`scipy.integrate.romberg`(*function*, *a*, *b*, *args=()*, *tol=1.48e-08*, *rtol=1.48e-08*, *show=False*, *divmax=10*, *vec_func=False*)

Romberg integration of a callable function or method.

Returns the integral of *function* (a function of one variable) over the interval (*a*, *b*).

If *show* is 1, the triangular array of the intermediate results will be printed. If *vec_func* is True (default is False), then *function* is assumed to support vector arguments.

Parameters

function : callable
: Function to be integrated.

a : float
    Lower limit of integration.
b : float
    Upper limit of integration.

Returns **results** : float
    Result of the integration.

Other Parameters

args : tuple, optional
    Extra arguments to pass to function. Each element of `args` will be passed as a single argument to *func*. Default is to pass no extra arguments.
tol, rtol : float, optional
    The desired absolute and relative tolerances. Defaults are 1.48e-8.
show : bool, optional
    Whether to print the results. Default is False.
divmax : int, optional
    Maximum order of extrapolation. Default is 10.
vec_func : bool, optional
    Whether *func* handles arrays as arguments (i.e whether it is a "vector" function). Default is False.

See also:

`fixed_quad` Fixed-order Gaussian quadrature.

`quad` Adaptive quadrature using QUADPACK.

`dblquad` Double integrals.

`tplquad` Triple integrals.

`romb` Integrators for sampled data.

`simps` Integrators for sampled data.

`cumtrapz` Cumulative integration for sampled data.

`ode` ODE integrator.

`odeint` ODE integrator.

References

[R45]

Examples

Integrate a gaussian from 0 to 1 and compare to the error function.

```
>>> from scipy import integrate
>>> from scipy.special import erf
>>> gaussian = lambda x: 1/np.sqrt(np.pi) * np.exp(-x**2)
>>> result = integrate.romberg(gaussian, 0, 1, show=True)
Romberg integration of <function vfunc at ...> from [0, 1]
```

```
Steps  StepSize  Results
    1  1.000000  0.385872
    2  0.500000  0.412631  0.421551
    4  0.250000  0.419184  0.421368  0.421356
    8  0.125000  0.420810  0.421352  0.421350  0.421350
```

```
  16  0.062500  0.421215  0.421350  0.421350  0.421350  0.421350
  32  0.031250  0.421317  0.421350  0.421350  0.421350  0.421350  0.421350
```

The final result is 0.421350396475 after 33 function evaluations.

```
>>> print("%g %g" % (2*result, erf(1)))
0.842701 0.842701
```

`scipy.integrate.quad_explain`(*output=<open file '<stdout>', mode 'w'>*)

Print extra information about integrate.quad() parameters and returns.

Parameters **output** : instance with "write" method, optional
Information about `quad` is passed to `output.write()`. Default is `sys.stdout`.

Returns None

`scipy.integrate.newton_cotes`(*rn, equal=0*)

Return weights and error coefficient for Newton-Cotes integration.

Suppose we have (N+1) samples of f at the positions x_0, x_1, ..., x_N. Then an N-point Newton-Cotes formula for the integral between x_0 and x_N is:

$\int_{x_0}^{x_N} f(x)dx = \Delta x \sum_{i=0}^{N} a_i f(x_i) + B_N (\Delta x)^{N+2} f^{N+1}(\xi)$

where $\xi \in [x_0, x_N]$ and $\Delta x = \frac{x_N - x_0}{N}$ is the average samples spacing.

If the samples are equally-spaced and N is even, then the error term is $B_N (\Delta x)^{N+3} f^{N+2}(\xi)$.

Parameters **rn** : int
The integer order for equally-spaced data or the relative positions of the samples with the first sample at 0 and the last at N, where N+1 is the length of *rn*. N is the order of the Newton-Cotes integration.

equal : int, optional
Set to 1 to enforce equally spaced data.

Returns **an** : ndarray
1-D array of weights to apply to the function at the provided sample positions.

B : float
Error coefficient.

Notes

Normally, the Newton-Cotes rules are used on smaller integration regions and a composite rule is used to return the total integral.

exception `scipy.integrate.IntegrationWarning`

Warning on issues during integration.

5.6.2 Integrating functions, given fixed samples

trapz(y[, x, dx, axis])	Integrate along the given axis using the composite trapezoidal rule.
cumtrapz(y[, x, dx, axis, initial])	Cumulatively integrate y(x) using the composite trapezoidal rule.
simps(y[, x, dx, axis, even])	Integrate y(x) using samples along the given axis and the composite Simpson's rule.
romb(y[, dx, axis, show])	Romberg integration using samples of a function.

`scipy.integrate.trapz(y, x=None, dx=1.0, axis=-1)`

Integrate along the given axis using the composite trapezoidal rule.

Integrate *y* (*x*) along given axis.

Parameters

y : array_like
: Input array to integrate.

x : array_like, optional
: The sample points corresponding to the *y* values. If *x* is None, the sample points are assumed to be evenly spaced *dx* apart. The default is None.

dx : scalar, optional
: The spacing between sample points when *x* is None. The default is 1.

axis : int, optional
: The axis along which to integrate.

Returns

trapz : float
: Definite integral as approximated by trapezoidal rule.

See also:

sum, cumsum

Notes

Image *[R51]* illustrates trapezoidal rule – y-axis locations of points will be taken from *y* array, by default x-axis distances between points will be 1.0, alternatively they can be provided with *x* array or with *dx* scalar. Return value will be equal to combined area under the red lines.

References

[R50], *[R51]*

Examples

```
>>> np.trapz([1,2,3])
4.0
>>> np.trapz([1,2,3], x=[4,6,8])
8.0
>>> np.trapz([1,2,3], dx=2)
8.0
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.trapz(a, axis=0)
array([ 1.5,  2.5,  3.5])
>>> np.trapz(a, axis=1)
array([ 2.,  8.])
```

`scipy.integrate.cumtrapz(y, x=None, dx=1.0, axis=-1, initial=None)`

Cumulatively integrate y(x) using the composite trapezoidal rule.

Parameters

y : array_like
: Values to integrate.

x : array_like, optional
: The coordinate to integrate along. If None (default), use spacing *dx* between consecutive elements in *y*.

dx : float, optional
: Spacing between elements of *y*. Only used if *x* is None.

axis : int, optional

Specifies the axis to cumulate. Default is -1 (last axis).

initial : scalar, optional
If given, uses this value as the first value in the returned result. Typically this value should be 0. Default is None, which means no value at `x[0]` is returned and *res* has one element less than *y* along the axis of integration.

Returns **res** : ndarray
The result of cumulative integration of *y* along *axis*. If *initial* is None, the shape is such that the axis of integration has one less value than *y*. If *initial* is given, the shape is equal to that of *y*.

See also:

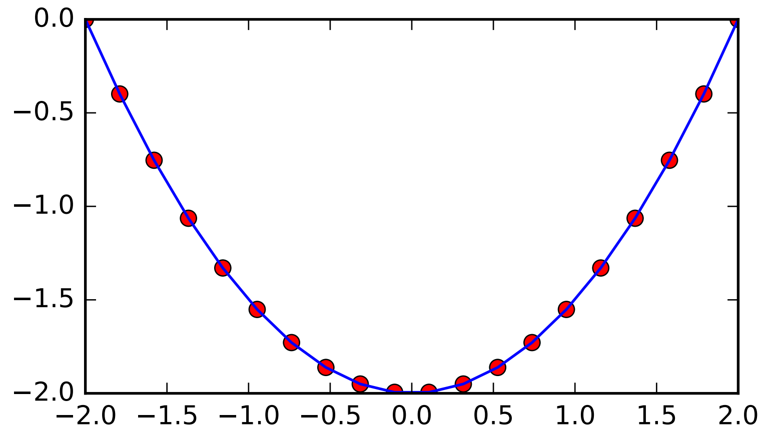
`numpy.cumsum`, `numpy.cumprod`

quad	adaptive quadrature using QUADPACK
romberg	adaptive Romberg quadrature
quadrature	adaptive Gaussian quadrature
fixed_quad	fixed-order Gaussian quadrature
dblquad	double integrals
tplquad	triple integrals
romb	integrators for sampled data
ode	ODE integrators
odeint	ODE integrators

Examples

```
>>> from scipy import integrate
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-2, 2, num=20)
>>> y = x
>>> y_int = integrate.cumtrapz(y, x, initial=0)
>>> plt.plot(x, y_int, 'ro', x, y[0] + 0.5 * x**2, 'b-')
>>> plt.show()
```



`scipy.integrate.`**`simps`**(*y*, *x=None*, *dx=1*, *axis=-1*, *even='avg'*)

Integrate y(x) using samples along the given axis and the composite Simpson's rule. If x is None, spacing of dx is assumed.

If there are an even number of samples, N, then there are an odd number of intervals (N-1), but Simpson's rule requires an even number of intervals. The parameter 'even' controls how this is handled.

Parameters

y : array_like
    Array to be integrated.

x : array_like, optional
    If given, the points at which *y* is sampled.

dx : int, optional
    Spacing of integration points along axis of *y*. Only used when *x* is None. Default is 1.

axis : int, optional
    Axis along which to integrate. Default is the last axis.

even : str {'avg', 'first', 'last'}, optional

'avg' [Average two results:1) use the first N-2 intervals with] a trapezoidal rule on the last interval and 2) use the last N-2 intervals with a trapezoidal rule on the first interval.

'first' [Use Simpson's rule for the first N-2 intervals with] a trapezoidal rule on the last interval.

'last' [Use Simpson's rule for the last N-2 intervals with a] trapezoidal rule on the first interval.

See also:

`quad` adaptive quadrature using QUADPACK

`romberg` adaptive Romberg quadrature

`quadrature` adaptive Gaussian quadrature

`fixed_quad` fixed-order Gaussian quadrature

`dblquad` double integrals

`tplquad` triple integrals

`romb` integrators for sampled data

cumtrapz cumulative integration for sampled data

ode ODE integrators

odeint ODE integrators

Notes

For an odd number of samples that are equally spaced the result is exact if the function is a polynomial of order 3 or less. If the samples are not equally spaced, then the result is exact only if the function is a polynomial of order 2 or less.

`scipy.integrate.romb`(*y*, *dx=1.0*, *axis=-1*, *show=False*)

Romberg integration using samples of a function.

Parameters **y** : array_like

A vector of `2**k + 1` equally-spaced samples of a function.

dx : float, optional

The sample spacing. Default is 1.

axis : int, optional

The axis along which to integrate. Default is -1 (last axis).

show : bool, optional

When *y* is a single 1-D array, then if this argument is True print the table showing Richardson extrapolation from the samples. Default is False.

Returns **romb** : ndarray

The integrated result for *axis*.

See also:

quad adaptive quadrature using QUADPACK

romberg adaptive Romberg quadrature

quadrature adaptive Gaussian quadrature

fixed_quad fixed-order Gaussian quadrature

dblquad double integrals

tplquad triple integrals

simps integrators for sampled data

cumtrapz cumulative integration for sampled data

ode ODE integrators

odeint ODE integrators

See also:

scipy.special for orthogonal polynomials (special) for Gaussian quadrature roots and weights for other weighting factors and regions.

5.6.3 Integrators of ODE systems

odeint(func, y0, t[, args, Dfun, col_deriv, ...])	Integrate a system of ordinary differential equations.
ode(f[, jac])	A generic interface class to numeric integrators.
complex_ode(f[, jac])	A wrapper of ode for complex systems.

Continued on next page

Table 5.20 – continued from previous page

solve_bvp(fun, bc, x, y[, p, S, fun_jac, ...])	Solve a boundary-value problem for a system of ODEs.

`scipy.integrate.`**`odeint`** (*func*, *y0*, *t*, *args=()*, *Dfun=None*, *col_deriv=0*, *full_output=0*, *ml=None*, *mu=None*, *rtol=None*, *atol=None*, *tcrit=None*, *h0=0.0*, *hmax=0.0*, *hmin=0.0*, *ixpr=0*, *mxstep=0*, *mxhnil=0*, *mxordn=12*, *mxords=5*, *printmessg=0*)

Integrate a system of ordinary differential equations.

Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff or non-stiff systems of first order ode-s:

```
dy/dt = func(y, t0, ...)
```

where y can be a vector.

Note: The first two arguments of `func(y, t0, ...)` are in the opposite order of the arguments in the system definition function used by the *scipy.integrate.ode* class.

Parameters

func : callable(y, t0, ...)
Computes the derivative of y at t0.

y0 : array
Initial condition on y (can be a vector).

t : array
A sequence of time points for which to solve for y. The initial value point should be the first element of this sequence.

args : tuple, optional
Extra arguments to pass to function.

Dfun : callable(y, t0, ...)
Gradient (Jacobian) of *func*.

col_deriv : bool, optional
True if *Dfun* defines derivatives down columns (faster), otherwise *Dfun* should define derivatives across rows.

full_output : bool, optional
True if to return a dictionary of optional outputs as the second output

printmessg : bool, optional
Whether to print the convergence message

Returns

y : array, shape (len(t), len(y0))
Array containing the value of y for each desired time in t, with the initial value *y0* in the first row.

infodict : dict, only returned if full_output == True
Dictionary containing additional output information

key	meaning
'hu'	vector of step sizes successfully used for each time step.
'tcur'	vector with the value of t reached for each time step. (will always be at least as large as the input times).
'tolsf'	vector of tolerance scale factors, greater than 1.0, computed when a request for too much accuracy was detected.
'tsw'	value of t at the time of the last method switch (given for each time step)
'nst'	cumulative number of time steps
'nfe'	cumulative number of function evaluations for each time step
'nje'	cumulative number of jacobian evaluations for each time step
'nqu'	a vector of method orders for each successful step.
'imxer'	index of the component of largest magnitude in the weighted local error vector (e / ewt) on an error return, -1 otherwise.
'lenrw'	the length of the double work array required.
'leniw'	the length of integer work array required.
'mused'	a vector of method indicators for each successful time step: 1: adams (nonstiff), 2: bdf (stiff)

Other Parameters

ml, mu : int, optional
If either of these are not None or non-negative, then the Jacobian is assumed to be banded. These give the number of lower and upper non-zero diagonals in this banded matrix. For the banded case, *Dfun* should return a matrix whose rows contain the non-zero bands (starting with the lowest diagonal). Thus, the return matrix *jac* from *Dfun* should have shape `(ml + mu + 1, len(y0))` when `ml >=0` or `mu >=0`. The data in *jac* must be stored such that `jac[i - j + mu, j]` holds the derivative of the *i'th equation with respect to the 'j'th state variable. If 'col_deriv* is True, the transpose of this *jac* must be returned.

rtol, atol : float, optional
The input parameters *rtol* and *atol* determine the error control performed by the solver. The solver will control the vector, e, of estimated local errors in y, according to an inequality of the form `max-norm of (e / ewt) <= 1`, where ewt is a vector of positive error weights computed as `ewt = rtol * abs(y) + atol`. rtol and atol can be either vectors the same length as y or scalars. Defaults to 1.49012e-8.

tcrit : ndarray, optional
Vector of critical points (e.g. singularities) where integration care should be taken.

h0 : float, (0: solver-determined), optional
The step size to be attempted on the first step.

hmax : float, (0: solver-determined), optional
The maximum absolute step size allowed.

hmin : float, (0: solver-determined), optional
The minimum absolute step size allowed.

ixpr : bool, optional
Whether to generate extra printing at method switches.

mxstep : int, (0: solver-determined), optional
Maximum number of (internally defined) steps allowed for each integration point in t.

mxhnil : int, (0: solver-determined), optional
Maximum number of messages printed.

mxordn : int, (0: solver-determined), optional
Maximum order to be allowed for the non-stiff (Adams) method.

mxords : int, (0: solver-determined), optional
Maximum order to be allowed for the stiff (BDF) method.

See also:

ode a more object-oriented integrator based on VODE.

quad for finding the area under a curve.

Examples

The second order differential equation for the angle *theta* of a pendulum acted on by gravity with friction can be written:

```
theta''(t) + b*theta'(t) + c*sin(theta(t)) = 0
```

where *b* and *c* are positive constants, and a prime (') denotes a derivative. To solve this equation with `odeint`, we must first convert it to a system of first order equations. By defining the angular velocity `omega(t) = theta'(t)`, we obtain the system:

```
theta'(t) = omega(t)
omega'(t) = -b*omega(t) - c*sin(theta(t))
```

Let *y* be the vector [*theta*, *omega*]. We implement this system in python as:

```
>>> def pend(y, t, b, c):
...     theta, omega = y
...     dydt = [omega, -b*omega - c*np.sin(theta)]
...     return dydt
...
```

We assume the constants are *b* = 0.25 and *c* = 5.0:

```
>>> b = 0.25
>>> c = 5.0
```

For initial conditions, we assume the pendulum is nearly vertical with *theta(0)* = *pi* - 0.1, and it initially at rest, so *omega(0)* = 0. Then the vector of initial conditions is

```
>>> y0 = [np.pi - 0.1, 0.0]
```

We generate a solution 101 evenly spaced samples in the interval 0 <= *t* <= 10. So our array of times is:

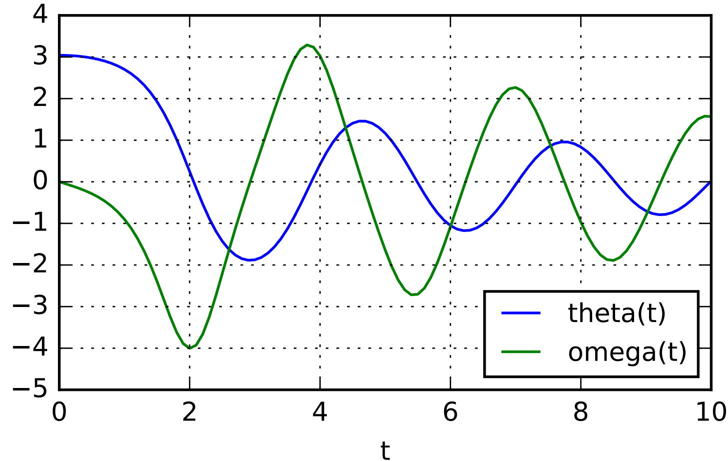
```
>>> t = np.linspace(0, 10, 101)
```

Call `odeint` to generate the solution. To pass the parameters *b* and *c* to *pend*, we give them to `odeint` using the `args` argument.

```
>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))
```

The solution is an array with shape (101, 2). The first column is *theta(t)*, and the second is *omega(t)*. The following code plots both components.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, sol[:, 0], 'b', label='theta(t)')
>>> plt.plot(t, sol[:, 1], 'g', label='omega(t)')
>>> plt.legend(loc='best')
>>> plt.xlabel('t')
>>> plt.grid()
>>> plt.show()
```



class `scipy.integrate.ode` (*f*, *jac=None*)

A generic interface class to numeric integrators.

Solve an equation system $y'(t) = f(t, y)$ with (optional) `jac = df/dy`.

Note: The first two arguments of `f(t, y, ...)` are in the opposite order of the arguments in the system definition function used by *scipy.integrate.odeint*.

Parameters

f : callable `f(t, y, *f_args)`
: Right-hand side of the differential equation. t is a scalar, `y.shape == (n,)`. `f_args` is set by calling `set_f_params(*args)`. f should return a scalar, array or list (not a tuple).

jac : callable `jac(t, y, *jac_args)`, optional
: Jacobian of the right-hand side, `jac[i,j] = d f[i] / d y[j]`. `jac_args` is set by calling `set_jac_params(*args)`.

See also:

odeint	an integrator with a simpler interface based on lsoda from ODEPACK
quad	for finding the area under a curve

Notes

Available integrators are listed below. They can be selected using the *set_integrator* method.

"vode"

Real-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
Source: http://www.netlib.org/ode/vode.f

> **Warning:** This integrator is not re-entrant. You cannot have two *ode* instances using the "vode" integrator at the same time.

This integrator accepts the following parameters in *set_integrator* method of the *ode* class:

- atol : float or sequence absolute tolerance for solution
- rtol : float or sequence relative tolerance for solution

- lband : None or int
- uband : None or int Jacobian band width, jac[i,j] != 0 for i-lband <= j <= i+uband. Setting these requires your jac routine to return the jacobian in packed format, jac_packed[i-j+uband, j] = jac[i,j]. The dimension of the matrix must be (lband+uband+1, len(y)).
- method: 'adams' or 'bdf' Which solver to use, Adams (non-stiff) or BDF (stiff)
- with_jacobian : bool This option is only considered when the user has not supplied a Jacobian function and has not indicated (by setting either band) that the Jacobian is banded. In this case, *with_jacobian* specifies whether the iteration method of the ODE solver's correction step is chord iteration with an internally generated full Jacobian or functional iteration with no Jacobian.
- nsteps : int Maximum number of (internally defined) steps allowed during one call to the solver.
- first_step : float
- min_step : float
- max_step : float Limits for the step sizes used by the integrator.
- order : int Maximum order used by the integrator, order <= 12 for Adams, <= 5 for BDF.

"zvode"

Complex-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
Source: http://www.netlib.org/ode/zvode.f

Warning: This integrator is not re-entrant. You cannot have two `ode` instances using the "zvode" integrator at the same time.

This integrator accepts the same parameters in `set_integrator` as the "vode" solver.

Note: When using ZVODE for a stiff system, it should only be used for the case in which the function f is analytic, that is, when each f(i) is an analytic function of each y(j). Analyticity means that the partial derivative df(i)/dy(j) is a unique complex number, and this fact is critical in the way ZVODE solves the dense or banded linear systems that arise in the stiff case. For a complex stiff ODE system in which f is not analytic, ZVODE is likely to have convergence failures, and for this problem one should instead use DVODE on the equivalent real system (in the real and imaginary parts of y).

"lsoda"

Real-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides automatic method switching between implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
Source: http://www.netlib.org/odepack

Warning: This integrator is not re-entrant. You cannot have two `ode` instances using the "lsoda" integrator at the same time.

This integrator accepts the following parameters in `set_integrator` method of the `ode` class:

- atol : float or sequence absolute tolerance for solution
- rtol : float or sequence relative tolerance for solution
- lband : None or int
- uband : None or int Jacobian band width, jac[i,j] != 0 for i-lband <= j <= i+uband. Setting these requires your jac routine to return the jacobian in packed format, jac_packed[i-j+uband, j] = jac[i,j].
- with_jacobian : bool *Not used.*
- nsteps : int Maximum number of (internally defined) steps allowed during one call to the solver.
- first_step : float
- min_step : float

- max_step : float Limits for the step sizes used by the integrator.
- max_order_ns : int Maximum order used in the nonstiff case (default 12).
- max_order_s : int Maximum order used in the stiff case (default 5).
- max_hnil : int Maximum number of messages reporting too small step size (t + h = t) (default 0)
- ixpr : int Whether to generate extra printing at method switches (default False).

"dopri5"

This is an explicit runge-kutta method of order (4)5 due to Dormand & Prince (with stepsize control and dense output).
Authors:

E. Hairer and G. Wanner Universite de Geneve, Dept. de Mathematiques CH-1211 Geneve 24, Switzerland e-mail: ernst.hairer@math.unige.ch, gerhard.wanner@math.unige.ch

This code is described in *[HNW93]*.
This integrator accepts the following parameters in set_integrator() method of the ode class:

- atol : float or sequence absolute tolerance for solution
- rtol : float or sequence relative tolerance for solution
- nsteps : int Maximum number of (internally defined) steps allowed during one call to the solver.
- first_step : float
- max_step : float
- safety : float Safety factor on new step selection (default 0.9)
- ifactor : float
- dfactor : float Maximum factor to increase/decrease step size by in one step
- beta : float Beta parameter for stabilised step size control.
- verbosity : int Switch for printing messages (< 0 for no messages).

"dop853"

This is an explicit runge-kutta method of order 8(5,3) due to Dormand & Prince (with stepsize control and dense output).
Options and references the same as "dopri5".

References

[HNW93]

Examples

A problem to integrate and the corresponding jacobian:

```
>>> from scipy.integrate import ode
>>>
>>> y0, t0 = [1.0j, 2.0], 0
>>>
>>> def f(t, y, arg1):
...     return [1j*arg1*y[0] + y[1], -arg1*y[1]**2]
>>> def jac(t, y, arg1):
...     return [[1j*arg1, 1], [0, -arg1*2*y[1]]]
```

The integration:

```
>>> r = ode(f, jac).set_integrator('zvode', method='bdf')
>>> r.set_initial_value(y0, t0).set_f_params(2.0).set_jac_params(2.0)
>>> t1 = 10
>>> dt = 1
>>> while r.successful() and r.t < t1:
...     print(r.t+dt, r.integrate(r.t+dt))
(1, array([-0.71038232+0.23749653j,  0.40000271+0.j        ]))
(2.0, array([ 0.19098503-0.52359246j,  0.22222356+0.j        ]))
```

```
(3.0, array([ 0.47153208+0.52701229j,  0.15384681+0.j        ]))
(4.0, array([-0.61905937+0.30726255j,  0.11764744+0.j        ]))
(5.0, array([ 0.02340997-0.61418799j,  0.09523835+0.j        ]))
(6.0, array([ 0.58643071+0.339819j,  0.08000018+0.j      ]))
(7.0, array([-0.52070105+0.44525141j,  0.06896565+0.j        ]))
(8.0, array([-0.15986733-0.61234476j,  0.06060616+0.j        ]))
(9.0, array([ 0.64850462+0.15048982j,  0.05405414+0.j        ]))
(10.0, array([-0.38404699+0.56382299j,  0.04878055+0.j        ]))
```

Attributes

t	(float) Current time.
y	(ndarray) Current variable values.

Methods

integrate(t[, step, relax])	Find y=y(t), set y as an initial condition, and return y.
set_f_params(*args)	Set extra parameters for user-supplied function f.
set_initial_value(y[, t])	Set initial conditions y(t) = y.
set_integrator(name, **integrator_params)	Set integrator by name.
set_jac_params(*args)	Set extra parameters for user-supplied function jac.
set_solout(solout)	Set callable to be called at every successful integration step.
successful()	Check if integration was successful.

ode.**integrate**(*t*, *step=0*, *relax=0*)
Find y=y(t), set y as an initial condition, and return y.

ode.**set_f_params**(**args*)
Set extra parameters for user-supplied function f.

ode.**set_initial_value**(*y*, *t=0.0*)
Set initial conditions y(t) = y.

ode.**set_integrator**(*name*, ***integrator_params*)
Set integrator by name.

Parameters **name** : str
Name of the integrator.
integrator_params
Additional parameters for the integrator.

ode.**set_jac_params**(**args*)
Set extra parameters for user-supplied function jac.

ode.**set_solout**(*solout*)
Set callable to be called at every successful integration step.

Parameters **solout** : callable
`solout(t, y)` is called at each internal integrator step, t is a scalar providing the current independent position y is the current soloution `y.shape == (n,)` solout should return -1 to stop integration otherwise it should return None or 0

ode.**successful**()
Check if integration was successful.

class scipy.integrate.**complex_ode**(*f*, *jac=None*)

A wrapper of ode for complex systems.

This functions similarly as *ode*, but re-maps a complex-valued equation system to a real-valued one before using the integrators.

Parameters **f** : callable f(t, y, *f_args)

Rhs of the equation. t is a scalar, y.shape == (n,). f_args is set by calling set_f_params(*args).

jac : callable jac(t, y, *jac_args)

Jacobian of the rhs, jac[i,j] = d f[i] / d y[j]. jac_args is set by calling set_f_params(*args).

Examples

For usage examples, see *ode*.

Attributes

t	(float) Current time.
y	(ndarray) Current variable values.

Methods

integrate(t[, step, relax])	Find y=y(t), set y as an initial condition, and return y.
set_f_params(*args)	Set extra parameters for user-supplied function f.
set_initial_value(y[, t])	Set initial conditions y(t) = y.
set_integrator(name, **integrator_params)	Set integrator by name.
set_jac_params(*args)	Set extra parameters for user-supplied function jac.
set_solout(solout)	Set callable to be called at every successful integration step.
successful()	Check if integration was successful.

complex_ode.**integrate**(*t*, *step=0*, *relax=0*)

Find y=y(t), set y as an initial condition, and return y.

complex_ode.**set_f_params**(*args)

Set extra parameters for user-supplied function f.

complex_ode.**set_initial_value**(*y*, *t=0.0*)

Set initial conditions y(t) = y.

complex_ode.**set_integrator**(*name*, ***integrator_params*)

Set integrator by name.

Parameters **name** : str

Name of the integrator

integrator_params

Additional parameters for the integrator.

complex_ode.**set_jac_params**(*args)

Set extra parameters for user-supplied function jac.

complex_ode.**set_solout**(*solout*)

Set callable to be called at every successful integration step.

Parameters **solout** : callable

`solout(t, y)` is called at each internal integrator step, t is a scalar providing the current independent position y is the current soloution `y.shape == (n,)` solout should return -1 to stop integration otherwise it should return None or 0

`complex_ode.`**`successful`**`()`

Check if integration was successful.

`scipy.integrate.`**`solve_bvp`**`(`*fun*, *bc*, *x*, *y*, *p=None*, *S=None*, *fun_jac=None*, *bc_jac=None*, *tol=0.001*, *max_nodes=1000*, *verbose=0*`)`

Solve a boundary-value problem for a system of ODEs.

This function numerically solves a first order system of ODEs subject to two-point boundary conditions:

```
dy / dx = f(x, y, p) + S * y / (x - a), a <= x <= b
bc(y(a), y(b), p) = 0
```

Here x is a 1-dimensional independent variable, y(x) is a n-dimensional vector-valued function and p is a k-dimensional vector of unknown parameters which is to be found along with y(x). For the problem to be determined there must be n + k boundary conditions, i.e. bc must be (n + k)-dimensional function.

The last singular term in the right-hand side of the system is optional. It is defined by an n-by-n matrix S, such that the solution must satisfy S y(a) = 0. This condition will be forced during iterations, so it must not contradict boundary conditions. See *[R47]* for the explanation how this term is handled when solving BVPs numerically.

Problems in a complex domain can be solved as well. In this case y and p are considered to be complex, and f and bc are assumed to be complex-valued functions, but x stays real. Note that f and bc must be complex differentiable (satisfy Cauchy-Riemann equations *[R49]*), otherwise you should rewrite your problem for real and imaginary parts separately. To solve a problem in a complex domain, pass an initial guess for y with a complex data type (see below).

Parameters

fun : callable

Right-hand side of the system. The calling signature is `fun(x, y)`, or `fun(x, y, p)` if parameters are present. All arguments are ndarray: `x` with shape (m,), `y` with shape (n, m), meaning that `y[:, i]` corresponds to `x[i]`, and `p` with shape (k,). The return value must be an array with shape (n, m) and with the same layout as `y`.

bc : callable

Function evaluating residuals of the boundary conditions. The calling signature is `bc(ya, yb)`, or `bc(ya, yb, p)` if parameters are present. All arguments are ndarray: `ya` and `yb` with shape (n,), and `p` with shape (k,). The return value must be an array with shape (n + k,).

x : array_like, shape (m,)

Initial mesh. Must be a strictly increasing sequence of real numbers with `x[0]=a` and `x[-1]=b`.

y : array_like, shape (n, m)

Initial guess for the function values at the mesh nodes, i-th column corresponds to `x[i]`. For problems in a complex domain pass *y* with a complex data type (even if the initial guess is purely real).

p : array_like with shape (k,) or None, optional

Initial guess for the unknown parameters. If None (default), it is assumed that the problem doesn't depend on any parameters.

S : array_like with shape (n, n) or None

Matrix defining the singular term. If None (default), the problem is solved without the singular term.

fun_jac : callable or None, optional

Function computing derivatives of f with respect to y and p. The calling signature is `fun_jac(x, y)`, or `fun_jac(x, y, p)` if parameters are present. The return must contain 1 or 2 elements in the following order:

- df_dy : array_like with shape (n, n, m) where an element (i, j, q) equals to d f_i(x_q, y_q, p) / d (y_q)_j.
- df_dp : array_like with shape (n, k, m) where an element (i, j, q) equals to d f_i(x_q, y_q, p) / d p_j.

Here q numbers nodes at which x and y are defined, whereas i and j number vector components. If the problem is solved without unknown parameters df_dp should not be returned.

If *fun_jac* is None (default), the derivatives will be estimated by the forward finite differences.

bc_jac : callable or None, optional

Function computing derivatives of bc with respect to ya, yb and p. The calling signature is `bc_jac(ya, yb)`, or `bc_jac(ya, yb, p)` if parameters are present. The return must contain 2 or 3 elements in the following order:

- dbc_dya : array_like with shape (n, n) where an element (i, j) equals to d bc_i(ya, yb, p) / d ya_j.
- dbc_dyb : array_like with shape (n, n) where an element (i, j) equals to d bc_i(ya, yb, p) / d yb_j.
- dbc_dp : array_like with shape (n, k) where an element (i, j) equals to d bc_i(ya, yb, p) / d p_j.

If the problem is solved without unknown parameters dbc_dp should not be returned.

If *bc_jac* is None (default), the derivatives will be estimated by the forward finite differences.

tol : float, optional

Desired tolerance of the solution. If we define `r = y' - f(x, y)` where y is the found solution, then the solver tries to achieve on each mesh interval `norm(r / (1 + abs(f)) < tol`, where `norm` is estimated in a root mean squared sense (using a numerical quadrature formula). Default is 1e-3.

max_nodes : int, optional

Maximum allowed number of the mesh nodes. If exceeded, the algorithm terminates. Default is 1000.

verbose : {0, 1, 2}, optional

Level of algorithm's verbosity:

- 0 (default) : work silently.
- 1 : display a termination report.
- 2 : display progress during iterations.

Returns Bunch object with the following fields defined:

sol : PPoly

Found solution for y as *scipy.interpolate.PPoly* instance, a C1 continuous cubic spline.

p : ndarray or None, shape (k,)

Found parameters. None, if the parameters were not present in the problem.

x : ndarray, shape (m,)

Nodes of the final mesh.

y : ndarray, shape (n, m)

Solution values at the mesh nodes.

yp : ndarray, shape (n, m)

Solution derivatives at the mesh nodes.

rms_residuals : ndarray, shape (m - 1,)

RMS values of the relative residuals over each mesh interval (see the description of *tol* parameter).

niter : int

Number of completed iterations.

status : int

Reason for algorithm termination:

- 0: The algorithm converged to the desired accuracy.
- 1: The maximum number of mesh nodes is exceeded.
- 2: A singular Jacobian encountered when solving the collocation system.

message : string
Verbal description of the termination reason.

success : bool
True if the algorithm converged to the desired accuracy (`status=0`).

Notes

This function implements a 4-th order collocation algorithm with the control of residuals similar to *[R46]*. A collocation system is solved by a damped Newton method with an affine-invariant criterion function as described in *[R48]*.

Note that in *[R46]* integral residuals are defined without normalization by interval lengths. So their definition is different by a multiplier of h**0.5 (h is an interval length) from the definition used here.

New in version 0.18.0.

References

[R46], *[R47]*, *[R48]*, *[R49]*

Examples

In the first example we solve Bratu's problem:

```
y'' + k * exp(y) = 0
y(0) = y(1) = 0
```

for k = 1.

We rewrite the equation as a first order system and implement its right-hand side evaluation:

```
y1' = y2
y2' = -exp(y1)
```

```
>>> def fun(x, y):
...     return np.vstack((y[1], -np.exp(y[0])))
```

Implement evaluation of the boundary condition residuals:

```
>>> def bc(ya, yb):
...     return np.array([ya[0], yb[0]])
```

Define the initial mesh with 5 nodes:

```
>>> x = np.linspace(0, 1, 5)
```

This problem is known to have two solutions. To obtain both of them we use two different initial guesses for y. We denote them by subscripts a and b.

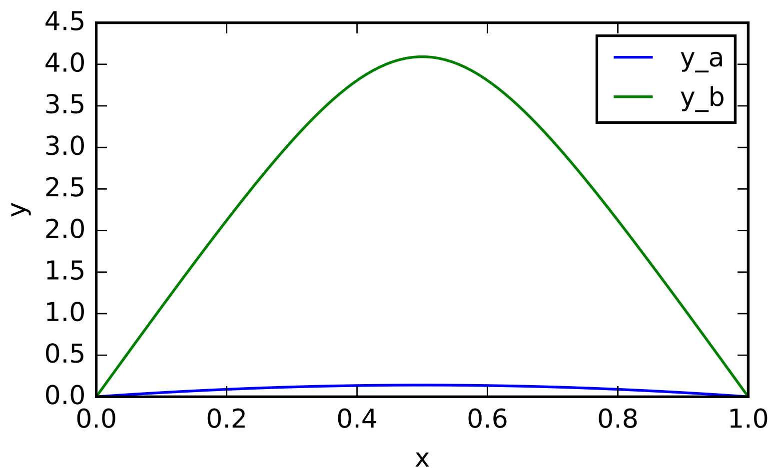
```
>>> y_a = np.zeros((2, x.size))
>>> y_b = np.zeros((2, x.size))
>>> y_b[0] = 3
```

Now we are ready to run the solver.

```
>>> from scipy.integrate import solve_bvp
>>> res_a = solve_bvp(fun, bc, x, y_a)
>>> res_b = solve_bvp(fun, bc, x, y_b)
```

Let's plot the two found solutions. We take an advantage of having the solution in a spline form to produce a smooth plot.

```
>>> x_plot = np.linspace(0, 1, 100)
>>> y_plot_a = res_a.sol(x_plot)[0]
>>> y_plot_b = res_b.sol(x_plot)[0]
>>> import matplotlib.pyplot as plt
>>> plt.plot(x_plot, y_plot_a, label='y_a')
>>> plt.plot(x_plot, y_plot_b, label='y_b')
>>> plt.legend()
>>> plt.xlabel("x")
>>> plt.ylabel("y")
>>> plt.show()
```



We see that the two solutions have similar shape, but differ in scale significantly.

In the second example we solve a simple Sturm-Liouville problem:

```
y'' + k**2 * y = 0
y(0) = y(1) = 0
```

It is known that a non-trivial solution y = A * sin(k * x) is possible for k = pi * n, where n is an integer. To establish the normalization constant A = 1 we add a boundary condition:

```
y'(0) = k
```

Again we rewrite our equation as a first order system and implement its right-hand side evaluation:

```
y1' = y2
y2' = -k**2 * y1
```

```
>>> def fun(x, y, p):
...     k = p[0]
```

```
...     return np.vstack((y[1], -k**2 * y[0]))
```

Note that parameters p are passed as a vector (with one element in our case).

Implement the boundary conditions:

```
>>> def bc(ya, yb, p):
...     k = p[0]
...     return np.array([ya[0], yb[0], ya[1] - k])
```

Setup the initial mesh and guess for y. We aim to find the solution for k = 2 * pi, to achieve that we set values of y to approximately follow sin(2 * pi * x):

```
>>> x = np.linspace(0, 1, 5)
>>> y = np.zeros((2, x.size))
>>> y[0, 1] = 1
>>> y[0, 3] = -1
```

Run the solver with 6 as an initial guess for k.

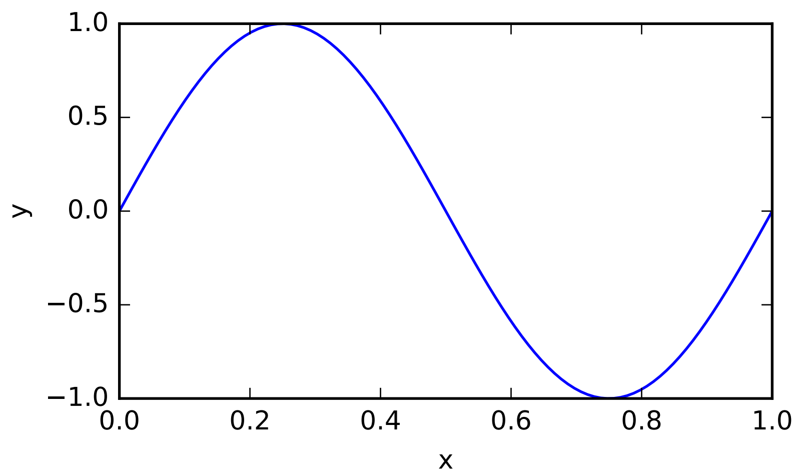
```
>>> sol = solve_bvp(fun, bc, x, y, p=[6])
```

We see that the found k is approximately correct:

```
>>> sol.p[0]
6.28329460046
```

And finally plot the solution to see the anticipated sinusoid:

```
>>> x_plot = np.linspace(0, 1, 100)
>>> y_plot = sol.sol(x_plot)[0]
>>> plt.plot(x_plot, y_plot)
>>> plt.xlabel("x")
>>> plt.ylabel("y")
>>> plt.show()
```



5.7 Interpolation (`scipy.interpolate`)

Sub-package for objects used in interpolation.

As listed below, this sub-package contains spline functions and classes, one-dimensional and multi-dimensional (univariate and multivariate) interpolation classes, Lagrange and Taylor polynomial interpolators, and wrappers for FITPACK and DFITPACK functions.

5.7.1 Univariate interpolation

interp1d(x, y[, kind, axis, copy, ...])	Interpolate a 1-D function.
BarycentricInterpolator(xi[, yi, axis])	The interpolating polynomial for a set of points
KroghInterpolator(xi, yi[, axis])	Interpolating polynomial for a set of points.
PchipInterpolator(x, y[, axis, extrapolate])	PCHIP 1-d monotonic cubic interpolation.
barycentric_interpolate(xi, yi, x[, axis])	Convenience function for polynomial interpolation.
krogh_interpolate(xi, yi, x[, der, axis])	Convenience function for polynomial interpolation.
pchip_interpolate(xi, yi, x[, der, axis])	Convenience function for pchip interpolation.
Akima1DInterpolator(x, y[, axis])	Akima interpolator
CubicSpline(x, y[, axis, bc_type, extrapolate])	Cubic spline data interpolator.
PPoly(c, x[, extrapolate, axis])	Piecewise polynomial in terms of coefficients and breakpoints
BPoly(c, x[, extrapolate, axis])	Piecewise polynomial in terms of coefficients and breakpoints.

class scipy.interpolate.**interp1d**(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=None*, *fill_value=nan*, *assume_sorted=False*)

Interpolate a 1-D function.

x and *y* are arrays of values used to approximate some function f: `y = f(x)`. This class returns a function whose call method uses interpolation to find the value of new points.

Note that calling *interp1d* with NaNs present in input values results in undefined behaviour.

Parameters

x : (N,) array_like
A 1-D array of real values.

y : (...,N,...) array_like
A N-D array of real values. The length of *y* along the interpolation axis must be equal to the length of *x*.

kind : str or int, optional
Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic' where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order) or as an integer specifying the order of the spline interpolator to use. Default is 'linear'.

axis : int, optional
Specifies the axis of *y* along which to interpolate. Interpolation defaults to the last axis of *y*.

copy : bool, optional
If True, the class makes internal copies of x and y. If False, references to *x* and *y* are used. The default is to copy.

bounds_error : bool, optional
If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of x (where extrapolation is necessary). If False, out of bounds values are assigned *fill_value*. By default, an error is raised unless *fill_value="extrapolate"*.

fill_value : array-like or (array-like, array_like) or "extrapolate", optional

- if a ndarray (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN. The array-like must broadcast properly to the dimensions of the non-interpolation axes.
- If a two-element tuple, then the first element is used as a fill value for `x_new < x[0]` and the second element is used for `x_new > x[-1]`. Anything that is not a 2-element tuple (e.g., list or ndarray, regardless of shape) is taken to be a single array-like argument meant to be used for both bounds as `below, above = fill_value, fill_value`.
  New in version 0.17.0.
- If "extrapolate", then points outside the data range will be extrapolated.
  New in version 0.17.0.

assume_sorted : bool, optional

If False, values of *x* can be in any order and they are sorted first. If True, *x* has to be an array of monotonically increasing values.

See also:

`splrep`, `splev`

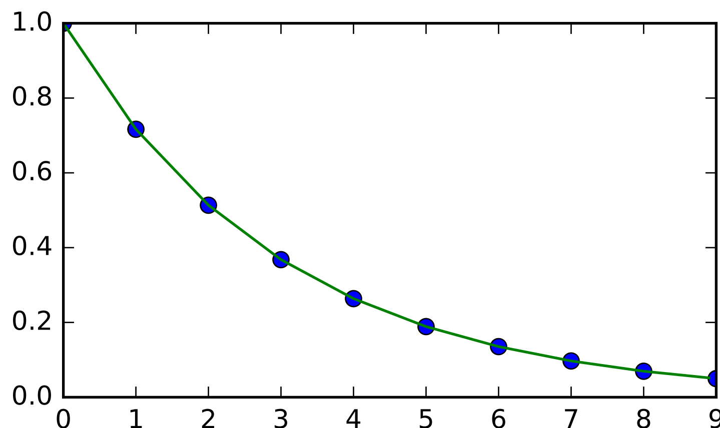
`UnivariateSpline` An object-oriented wrapper of the FITPACK routines.

`interp2d` 2-D interpolation

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
>>> x = np.arange(0, 10)
>>> y = np.exp(-x/3.0)
>>> f = interpolate.interp1d(x, y)
```

```
>>> xnew = np.arange(0, 9, 0.1)
>>> ynew = f(xnew)   # use interpolation function returned by `interp1d`
>>> plt.plot(x, y, 'o', xnew, ynew, '-')
>>> plt.show()
```



Attributes

fill_value

interp1d.**fill_value**

Methods

`__call__`(x)	Evaluate the interpolant

interp1d.**__call__**(*x*)
Evaluate the interpolant

Parameters **x** : array_like
Points to evaluate the interpolant at.

Returns **y** : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

class scipy.interpolate.**BarycentricInterpolator**(*xi*, *yi=None*, *axis=0*)
The interpolating polynomial for a set of points

Constructs a polynomial that passes through a given set of points. Allows evaluation of the polynomial, efficient changing of the y values to be interpolated, and updating by adding more x values. For reasons of numerical stability, this function does not compute the coefficients of the polynomial.

The values yi need to be provided before the function is evaluated, but none of the preprocessing depends on them, so rapid updates are possible.

Parameters **xi** : array_like
1-d array of x coordinates of the points the polynomial should pass through

yi : array_like, optional
The y coordinates of the points the polynomial should pass through. If None, the y values will be supplied later via the *set_y* method.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

Notes

This class uses a "barycentric interpolation" method that treats the problem as a special case of rational function interpolation. This algorithm is quite stable, numerically, but even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon.

Based on Berrut and Trefethen 2004, "Barycentric Lagrange Interpolation".

Methods

`__call__`(x)	Evaluate the interpolating polynomial at the points x
add_xi(xi[, yi])	Add more x values to the set to be interpolated
set_yi(yi[, axis])	Update the y values to be interpolated

BarycentricInterpolator.**__call__**(*x*)
Evaluate the interpolating polynomial at the points x

Parameters **x** : array_like
Points to evaluate the interpolant at.

Returns **y** : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Currently the code computes an outer product between x and the weights, that is, it constructs an intermediate array of size N by len(x), where N is the degree of the polynomial.

`BarycentricInterpolator.`**`add_xi`**(*xi*, *yi=None*)

Add more x values to the set to be interpolated

The barycentric interpolation algorithm allows easy updating by adding more points for the polynomial to pass through.

Parameters **xi** : array_like
The x coordinates of the points that the polynomial should pass through.

yi : array_like, optional
The y coordinates of the points the polynomial should pass through. Should have shape `(xi.size, R)`; if R > 1 then the polynomial is vector-valued. If *yi* is not given, the y values will be supplied later. *yi* should be given if and only if the interpolator has y values specified.

`BarycentricInterpolator.`**`set_yi`**(*yi*, *axis=None*)

Update the y values to be interpolated

The barycentric interpolation algorithm requires the calculation of weights, but these depend only on the xi. The yi can be changed at any time.

Parameters **yi** : array_like
The y coordinates of the points the polynomial should pass through. If None, the y values will be supplied later.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

class `scipy.interpolate.`**`KroghInterpolator`**(*xi*, *yi*, *axis=0*)

Interpolating polynomial for a set of points.

The polynomial passes through all the pairs (xi,yi). One may additionally specify a number of derivatives at each point xi; this is done by repeating the value xi and specifying the derivatives as successive yi values.

Allows evaluation of the polynomial and all its derivatives. For reasons of numerical stability, this function does not compute the coefficients of the polynomial, although they can be obtained by evaluating all the derivatives.

Parameters **xi** : array_like, length N
Known x-coordinates. Must be sorted in increasing order.

yi : array_like
Known y-coordinates. When an xi occurs two or more times in a row, the corresponding yi's represent derivative values.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

Notes

Be aware that the algorithms implemented here are not necessarily the most numerically stable known. Moreover, even in a world of exact computation, unless the x coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due

to the Runge phenomenon. In general, even with well-chosen x values, degrees higher than about thirty cause problems with numerical instability in this code.

Based on *[R60]*.

References

[R60]

Examples

To produce a polynomial that is zero at 0 and 1 and has derivative 2 at 0, call

```
>>> from scipy.interpolate import KroghInterpolator
>>> KroghInterpolator([0,0,1],[0,2,0])
```

This constructs the quadratic 2*X**2-2*X. The derivative condition is indicated by the repeated zero in the xi array; the corresponding yi values are 0, the function value, and 2, the derivative value.

For another example, given xi, yi, and a derivative ypi for each point, appropriate arrays can be constructed as:

```
>>> xi = np.linspace(0, 1, 5)
>>> yi, ypi = np.random.rand(2, 5)
>>> xi_k, yi_k = np.repeat(xi, 2), np.ravel(np.dstack((yi,ypi)))
>>> KroghInterpolator(xi_k, yi_k)
```

To produce a vector-valued polynomial, supply a higher-dimensional array for yi:

```
>>> KroghInterpolator([0,1],[[2,3],[4,5]])
```

This constructs a linear polynomial giving (2,3) at 0 and (4,5) at 1.

Methods

`__call__`(x)	Evaluate the interpolant
`derivative`(x[, der])	Evaluate one derivative of the polynomial at the point x
`derivatives`(x[, der])	Evaluate many derivatives of the polynomial at the point x

`KroghInterpolator.`**`__call__`**(*x*)

Evaluate the interpolant

Parameters **x** : array_like
Points to evaluate the interpolant at.

Returns **y** : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

`KroghInterpolator.`**`derivative`**(*x*, *der=1*)

Evaluate one derivative of the polynomial at the point x

Parameters **x** : array_like
Point or points at which to evaluate the derivatives

der : integer, optional
Which derivative to extract. This number includes the function value as 0th derivative.

Returns **d** : ndarray

Derivative interpolated at the x-points. Shape of d is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

This is computed by evaluating all derivatives up to the desired one (using self.derivatives()) and then discarding the rest.

`KroghInterpolator.`**`derivatives`**(*x*, *der=None*)

Evaluate many derivatives of the polynomial at the point x

Produce an array of all derivative values at the point x.

Parameters

x : array_like
Point or points at which to evaluate the derivatives

der : int or None, optional
How many derivatives to extract; None for all potentially nonzero derivatives (that is a number equal to the number of points). This number includes the function value as 0th derivative.

Returns

d : ndarray
Array with derivatives; d[j] contains the j-th derivative. Shape of d[j] is determined by replacing the interpolation axis in the original array with the shape of x.

Examples

```
>>> from scipy.interpolate import KroghInterpolator
>>> KroghInterpolator([0,0,0],[1,2,3]).derivatives(0)
array([1.0,2.0,3.0])
>>> KroghInterpolator([0,0,0],[1,2,3]).derivatives([0,0])
array([[1.0,1.0],
       [2.0,2.0],
       [3.0,3.0]])
```

class `scipy.interpolate.`**`PchipInterpolator`**(*x*, *y*, *axis=0*, *extrapolate=None*)

PCHIP 1-d monotonic cubic interpolation.

`x` and *y* are arrays of values used to approximate some function f, with `y = f(x)`. The interpolant uses monotonic cubic splines to find the value of new points. (PCHIP stands for Piecewise Cubic Hermite Interpolating Polynomial).

Parameters

x : ndarray
A 1-D array of monotonically increasing real values. `x` cannot include duplicate values (otherwise f is overspecified)

y : ndarray
A 1-D array of real values. *y*'s length along the interpolation axis must be equal to the length of `x`. If N-D array, use `axis` parameter to select correct axis.

axis : int, optional
Axis in the y array corresponding to the x-coordinate values.

extrapolate : bool, optional
Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

See also:

`Akima1DInterpolator`, *`CubicSpline`*, *`BPoly`*

Notes

The interpolator preserves monotonicity in the interpolation data and does not overshoot if the data is not smooth.

The first derivatives are guaranteed to be continuous, but the second derivatives may jump at x_k.

Determines the derivatives at the points x_k, f'_k, by using PCHIP algorithm *[R62]*.

Let $h_k = x_{k+1} - x_k$, and $d_k = (y_{k+1} - y_k)/h_k$ are the slopes at internal points x_k. If the signs of d_k and d_{k-1} are different or either of them equals zero, then $f'_k = 0$. Otherwise, it is given by the weighted harmonic mean

$$\frac{w_1 + w_2}{f'_k} = \frac{w_1}{d_{k-1}} + \frac{w_2}{d_k}$$

where $w_1 = 2h_k + h_{k-1}$ and $w_2 = h_k + 2h_{k-1}$.

The end slopes are set using a one-sided scheme *[R63]*.

References

[R62], *[R63]*

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative.
`derivative`([nu])	Construct a new piecewise polynomial representing the derivative.
`antiderivative`([nu])	Construct a new piecewise polynomial representing the antiderivative.
`roots`()	Return the roots of the interpolated function.

`PchipInterpolator.`**`__call__`**(*x*, *nu=0*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative.

Parameters

x : array_like
Points to evaluate the interpolant at.

nu : int, optional
Order of derivative to evaluate. Must be non-negative.

extrapolate : {bool, 'periodic', None}, optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns

y : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`PchipInterpolator.`**`derivative`**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

Parameters

nu : int, optional
Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns

bp : BPoly

Piecewise polynomial of order k - nu representing the derivative of this polynomial.

`PchipInterpolator.`**`antiderivative`**(*nu=1*)

Construct a new piecewise polynomial representing the antiderivative.

Parameters **nu** : int, optional
Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns **bp** : BPoly
Piecewise polynomial of order k + nu representing the antiderivative of this polynomial.

Notes

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`PchipInterpolator.`**`roots`**()

Return the roots of the interpolated function.

`scipy.interpolate.`**`barycentric_interpolate`**(*xi*, *yi*, *x*, *axis=0*)

Convenience function for polynomial interpolation.

Constructs a polynomial that passes through a given set of points, then evaluates the polynomial. For reasons of numerical stability, this function does not compute the coefficients of the polynomial.

This function uses a "barycentric interpolation" method that treats the problem as a special case of rational function interpolation. This algorithm is quite stable, numerically, but even in a world of exact computation, unless the *x* coordinates are chosen very carefully - Chebyshev zeros (e.g. cos(i*pi/n)) are a good choice - polynomial interpolation itself is a very ill-conditioned process due to the Runge phenomenon.

Parameters **xi** : array_like
1-d array of x coordinates of the points the polynomial should pass through

yi : array_like
The y coordinates of the points the polynomial should pass through.

x : scalar or array_like
Points to evaluate the interpolator at.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

Returns **y** : scalar or array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

See also:

`BarycentricInterpolator`

Notes

Construction of the interpolation weights is a relatively slow process. If you want to call this many times with the same xi (but possibly varying yi or x) you should use the class *`BarycentricInterpolator`*. This is what this function uses internally.

`scipy.interpolate.`**`krogh_interpolate`**(*xi*, *yi*, *x*, *der=0*, *axis=0*)

Convenience function for polynomial interpolation.

See *`KroghInterpolator`* for more details.

Parameters **xi** : array_like
Known x-coordinates.

yi : array_like
Known y-coordinates, of shape `(xi.size, R)`. Interpreted as vectors of length R, or scalars if R=1.

x : array_like
Point or points at which to evaluate the derivatives.

der : int or list, optional
How many derivatives to extract; None for all potentially nonzero derivatives (that is a number equal to the number of points), or a list of derivatives to extract. This number includes the function value as 0th derivative.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

Returns

d : ndarray
If the interpolator's values are R-dimensional then the returned array will be the number of derivatives by N by R. If *x* is a scalar, the middle dimension will be dropped; if the *yi* are scalars then the last dimension will be dropped.

See also:

KroghInterpolator

Notes

Construction of the interpolating polynomial is a relatively expensive process. If you want to evaluate it repeatedly consider using the class KroghInterpolator (which is what this function uses).

scipy.interpolate.**pchip_interpolate**(*xi*, *yi*, *x*, *der=0*, *axis=0*)

Convenience function for pchip interpolation. xi and yi are arrays of values used to approximate some function f, with `yi = f(xi)`. The interpolant uses monotonic cubic splines to find the value of new points x and the derivatives there.

See *PchipInterpolator* for details.

Parameters

xi : array_like
A sorted list of x-coordinates, of length N.

yi : array_like
A 1-D array of real values. *yi*'s length along the interpolation axis must be equal to the length of *xi*. If N-D array, use axis parameter to select correct axis.

x : scalar or array_like
Of length M.

der : int or list, optional
Derivatives to extract. The 0-th derivative can be included to return the function value.

axis : int, optional
Axis in the yi array corresponding to the x-coordinate values.

Returns

y : scalar or array_like
The result, of length R or length M or M by R,

See also:

PchipInterpolator

class scipy.interpolate.**Akima1DInterpolator**(*x*, *y*, *axis=0*)

Akima interpolator

Fit piecewise cubic polynomials, given vectors x and y. The interpolation method by Akima uses a continuously differentiable sub-spline built from piecewise cubic polynomials. The resultant curve passes through the given data points and will appear smooth and natural.

Parameters

x : ndarray, shape (m,)
1-D array of monotonically increasing real values.

y : ndarray, shape (m, ...)
N-D array of real values. The length of *y* along the first axis must be equal to the length of x.
axis : int, optional
Specifies the axis of *y* along which to interpolate. Interpolation defaults to the first axis of *y*.

See also:

PchipInterpolator, *CubicSpline*, *PPoly*

Notes

New in version 0.14.

Use only for precise data, as the fitted curve passes through the given points exactly. This routine is useful for plotting a pleasingly smooth curve through a few given points for purposes of plotting.

References

[1] A new method of interpolation and smooth curve fitting based
on local procedures. Hiroshi Akima, J. ACM, October 1970, 17(4), 589-602.

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative.
`derivative`([nu])	Construct a new piecewise polynomial representing the derivative.
`antiderivative`([nu])	Construct a new piecewise polynomial representing the antiderivative.
`roots`([discontinuity, extrapolate])	Find real roots of the the piecewise polynomial.

Akima1DInterpolator.**__call__**(*x*, *nu=0*, *extrapolate=None*)
Evaluate the piecewise polynomial or its derivative.

Parameters **x** : array_like
Points to evaluate the interpolant at.
nu : int, optional
Order of derivative to evaluate. Must be non-negative.
extrapolate : {bool, 'periodic', None}, optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns **y** : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

Akima1DInterpolator.**derivative**(*nu=1*)
Construct a new piecewise polynomial representing the derivative.

Parameters **nu** : int, optional

Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k - n representing the derivative of this polynomial.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`Akima1DInterpolator.`**`antiderivative`**`(`*nu=1*`)`

Construct a new piecewise polynomial representing the antiderivative.

Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

Parameters **nu** : int, optional

Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k + n representing the antiderivative of this polynomial.

Notes

The antiderivative returned by this function is continuous and continuously differentiable to order n-1, up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`Akima1DInterpolator.`**`roots`**`(`*discontinuity=True*, *extrapolate=None*`)`

Find real roots of the the piecewise polynomial.

Parameters **discontinuity** : bool, optional

Whether to report sign changes across discontinuities at breakpoints as roots.

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns **roots** : ndarray

Roots of the polynomial(s).

If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

See also:

`PPoly.solve`

class `scipy.interpolate.`**`CubicSpline`**`(`*x*, *y*, *axis=0*, *bc_type='not-a-knot'*, *extrapolate=None*`)`

Cubic spline data interpolator.

Interpolate data with a piecewise cubic polynomial which is twice continuously differentiable *[R58]*. The result is represented as a *`PPoly`* instance with breakpoints matching the given data.

Parameters **x** : array_like, shape (n,)

1-d array containing values of the independent variable. Values must be real, finite and in strictly increasing order.

y : array_like

Array containing values of the dependent variable. It can have arbitrary number of dimensions, but the length along `axis` (see below) must match the length of `x`. Values must be finite.

axis : int, optional

Axis along which *y* is assumed to be varying. Meaning that for `x[i]` the corresponding values are `np.take(y, i, axis=axis)`. Default is 0.

bc_type : string or 2-tuple, optional

Boundary condition type. Two additional equations, given by the boundary conditions, are required to determine all coefficients of polynomials on each segment *[R59]*.

If *bc_type* is a string, then the specified condition will be applied at both ends of a spline. Available conditions are:

- 'not-a-knot' (default): The first and second segment at a curve end are the same polynomial. It is a good default when there is no information on boundary conditions.
- 'periodic': The interpolated functions is assumed to be periodic of period `x[-1] - x[0]`. The first and last value of *y* must be identical: `y[0] == y[-1]`. This boundary condition will result in `y'[0] == y'[-1]` and `y''[0] == y''[-1]`.
- 'clamped': The first derivative at curves ends are zero. Assuming a 1D *y*, `bc_type=((1, 0.0), (1, 0.0))` is the same condition.
- 'natural': The second derivative at curve ends are zero. Assuming a 1D *y*, `bc_type=((2, 0.0), (2, 0.0))` is the same condition.

If *bc_type* is a 2-tuple, the first and the second value will be applied at the curve start and end respectively. The tuple values can be one of the previously mentioned strings (except 'periodic') or a tuple *(order, deriv_values)* allowing to specify arbitrary derivatives at curve ends:

- *order*: the derivative order, 1 or 2.
- *deriv_value*: array_like containing derivative values, shape must be the same as *y*, excluding `axis` dimension. For example, if *y* is 1D, then *deriv_value* must be a scalar. If *y* is 3D with the shape (n0, n1, n2) and axis=2, then *deriv_value* must be 2D and have the shape (n0, n1).

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), `extrapolate` is set to 'periodic' for `bc_type='periodic'` and to True otherwise.

See also:

Akima1DInterpolator, *PchipInterpolator*, *PPoly*

Notes

Parameters *bc_type* and `interpolate` work independently, i.e. the former controls only construction of a spline, and the latter only evaluation.

When a boundary condition is 'not-a-knot' and n = 2, it is replaced by a condition that the first derivative is equal to the linear interpolant slope. When both boundary conditions are 'not-a-knot' and n = 3, the solution is sought as a parabola passing through given points.

When 'not-a-knot' boundary conditions is applied to both ends, the resulting spline will be the same as returned by *splrep* (with `s=0`) and `InterpolatedUnivariateSpline`, but these two methods use a representation in B-spline basis.

New in version 0.18.0.

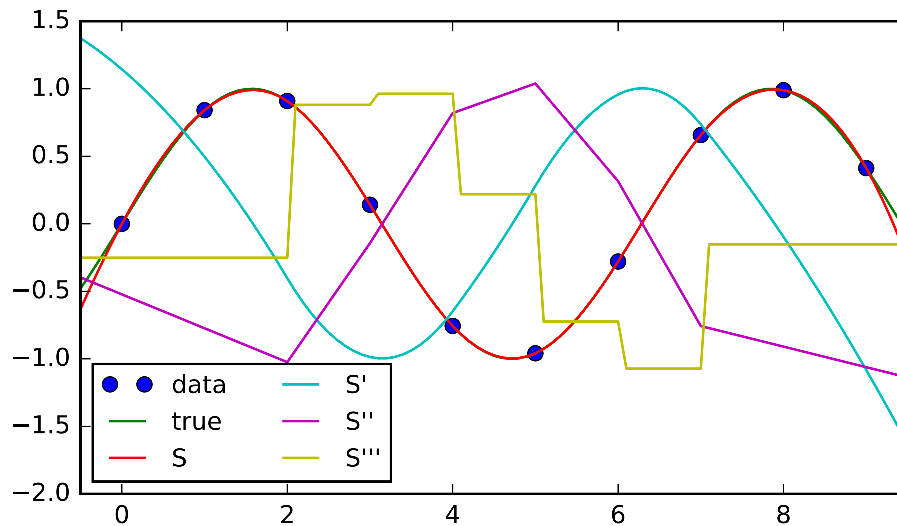
References

[R58], *[R59]*

Examples

In this example the cubic spline is used to interpolate a sampled sinusoid. You can see that the spline continuity property holds for the first and second derivatives and violates only for the third derivative.

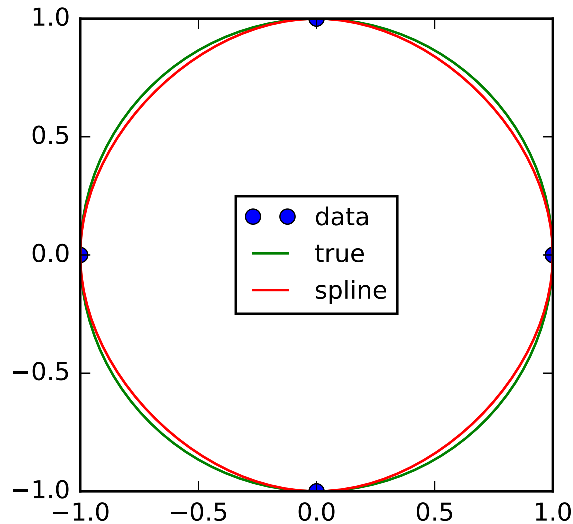
```
>>> from scipy.interpolate import CubicSpline
>>> import matplotlib.pyplot as plt
>>> x = np.arange(10)
>>> y = np.sin(x)
>>> cs = CubicSpline(x, y)
>>> xs = np.arange(-0.5, 9.6, 0.1)
>>> plt.figure(figsize=(6.5, 4))
>>> plt.plot(x, y, 'o', label='data')
>>> plt.plot(xs, np.sin(xs), label='true')
>>> plt.plot(xs, cs(xs), label="S")
>>> plt.plot(xs, cs(xs, 1), label="S'")
>>> plt.plot(xs, cs(xs, 2), label="S''")
>>> plt.plot(xs, cs(xs, 3), label="S'''")
>>> plt.xlim(-0.5, 9.5)
>>> plt.legend(loc='lower left', ncol=2)
>>> plt.show()
```



In the second example, the unit circle is interpolated with a spline. A periodic boundary condition is used. You can see that the first derivative values, ds/dx=0, ds/dy=1 at the periodic point (1, 0) are correctly computed. Note that a circle cannot be exactly represented by a cubic spline. To increase precision, more breakpoints would be required.

```
>>> theta = 2 * np.pi * np.linspace(0, 1, 5)
>>> y = np.c_[np.cos(theta), np.sin(theta)]
>>> cs = CubicSpline(theta, y, bc_type='periodic')
>>> print("ds/dx={:.1f} ds/dy={:.1f}".format(cs(0, 1)[0], cs(0, 1)[1]))
ds/dx=0.0 ds/dy=1.0
>>> xs = 2 * np.pi * np.linspace(0, 1, 100)
```

```
>>> plt.figure(figsize=(6.5, 4))
>>> plt.plot(y[:, 0], y[:, 1], 'o', label='data')
>>> plt.plot(np.cos(xs), np.sin(xs), label='true')
>>> plt.plot(cs(xs)[:, 0], cs(xs)[:, 1], label='spline')
>>> plt.axes().set_aspect('equal')
>>> plt.legend(loc='center')
>>> plt.show()
```



The third example is the interpolation of a polynomial y = x**3 on the interval 0 <= x<= 1. A cubic spline can represent this function exactly. To achieve that we need to specify values and first derivatives at endpoints of the interval. Note that y' = 3 * x**2 and thus y'(0) = 0 and y'(1) = 3.

```
>>> cs = CubicSpline([0, 1], [0, 1], bc_type=((1, 0), (1, 3)))
>>> x = np.linspace(0, 1)
>>> np.allclose(x**3, cs(x))
True
```

Attributes

x	(ndarray, shape (n,)) Breakpoints. The same `x` which was passed to the constructor.
c	(ndarray, shape (4, n-1, ...)) Coefficients of the polynomials on each segment. The trailing dimensions match the dimensions of *y*, excluding `axis`. For example, if *y* is 1-d, then `c[k, i]` is a coefficient for `(x-x[i])**(3-k)` on the segment between `x[i]` and `x[i+1]`.
axis	(int) Interpolation axis. The same `axis` which was passed to the constructor.

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative.
`derivative`([nu])	Construct a new piecewise polynomial representing the derivative.

Continued on next page

Table 5.30 – continued from previous page

antiderivative([nu])	Construct a new piecewise polynomial representing the antiderivative.
integrate(a, b[, extrapolate])	Compute a definite integral over a piecewise polynomial.
roots([discontinuity, extrapolate])	Find real roots of the the piecewise polynomial.

CubicSpline.**__call__**(*x*, *nu=0*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative.

Parameters **x** : array_like

Points to evaluate the interpolant at.

nu : int, optional

Order of derivative to evaluate. Must be non-negative.

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns **y** : array_like

Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

CubicSpline.**derivative**(*nu=1*)

Construct a new piecewise polynomial representing the derivative.

Parameters **nu** : int, optional

Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k - n representing the derivative of this polynomial.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

CubicSpline.**antiderivative**(*nu=1*)

Construct a new piecewise polynomial representing the antiderivative.

Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

Parameters **nu** : int, optional

Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k + n representing the antiderivative of this polynomial.

Notes

The antiderivative returned by this function is continuous and continuously differentiable to order n-1, up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`CubicSpline.`**`integrate`**(*a*, *b*, *extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

Parameters

a : float
    Lower integration bound

b : float
    Upper integration bound

extrapolate : {bool, 'periodic', None}, optional
    If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns

ig : array_like
    Definite integral of the piecewise polynomial over [a, b]

`CubicSpline.`**`roots`**(*discontinuity=True*, *extrapolate=None*)

Find real roots of the the piecewise polynomial.

Parameters

discontinuity : bool, optional
    Whether to report sign changes across discontinuities at breakpoints as roots.

extrapolate : {bool, 'periodic', None}, optional
    If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns

roots : ndarray
    Roots of the polynomial(s).
    If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

See also:

`PPoly.solve`

class `scipy.interpolate.`**`PPoly`**(*c*, *x*, *extrapolate=None*, *axis=0*)

Piecewise polynomial in terms of coefficients and breakpoints

The polynomial between `x[i]` and `x[i + 1]` is written in the local power basis:

```
S = sum(c[m, i] * (xp - x[i])**(k-m) for m in range(k+1))
```

where `k` is the degree of the polynomial.

Parameters

c : ndarray, shape (k, m, ...)
    Polynomial coefficients, order *k* and *m* intervals

x : ndarray, shape (m+1,)
    Polynomial breakpoints. Must be sorted in either increasing or decreasing order.

extrapolate : bool or 'periodic', optional
    If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

axis : int, optional
    Interpolation axis. Default is zero.

See also:

BPoly piecewise polynomials in the Bernstein basis

Notes

High-order polynomials in the power basis can be numerically unstable. Precision problems can start to appear for orders larger than 20-30.

Attributes

x	(ndarray) Breakpoints.
c	(ndarray) Coefficients of the polynomials. They are reshaped to a 3-dimensional array with the last dimension representing the trailing dimensions of the original coefficient array.
axis	(int) Interpolation axis.

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative.
`derivative`([nu])	Construct a new piecewise polynomial representing the derivative.
`antiderivative`([nu])	Construct a new piecewise polynomial representing the antiderivative.
`integrate`(a, b[, extrapolate])	Compute a definite integral over a piecewise polynomial.
`solve`([y, discontinuity, extrapolate])	Find real solutions of the the equation `pp(x) == y`.
`roots`([discontinuity, extrapolate])	Find real roots of the the piecewise polynomial.
`extend`(c, x[, right])	Add additional breakpoints and coefficients to the polynomial.
`from_spline`(tck[, extrapolate])	Construct a piecewise polynomial from a spline
`from_bernstein_basis`(bp[, extrapolate])	Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.
`construct_fast`(c, x[, extrapolate, axis])	Construct the piecewise polynomial without making checks.

`PPoly.`**`__call__`**(*x*, *nu=0*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative.

Parameters

x : array_like
Points to evaluate the interpolant at.

nu : int, optional
Order of derivative to evaluate. Must be non-negative.

extrapolate : {bool, 'periodic', None}, optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns

y : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`PPoly.derivative(nu=1)`

Construct a new piecewise polynomial representing the derivative.

Parameters **nu** : int, optional

Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k - n representing the derivative of this polynomial.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`PPoly.antiderivative(nu=1)`

Construct a new piecewise polynomial representing the antiderivative.

Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

Parameters **nu** : int, optional

Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns **pp** : PPoly

Piecewise polynomial of order k2 = k + n representing the antiderivative of this polynomial.

Notes

The antiderivative returned by this function is continuous and continuously differentiable to order n-1, up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`PPoly.integrate(a, b, extrapolate=None)`

Compute a definite integral over a piecewise polynomial.

Parameters **a** : float

Lower integration bound

b : float

Upper integration bound

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns **ig** : array_like

Definite integral of the piecewise polynomial over [a, b]

`PPoly.solve(y=0.0, discontinuity=True, extrapolate=None)`

Find real solutions of the the equation `pp(x) == y`.

Parameters **y** : float, optional

Right-hand side. Default is zero.

discontinuity : bool, optional

Whether to report sign changes across discontinuities at breakpoints as roots.

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns **roots** : ndarray
Roots of the polynomial(s).
If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

Notes

This routine works only on real-valued polynomials.

If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a `nan` value.

If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is True.

Examples

Finding roots of `[x**2 - 1, (x - 1)**2]` defined on intervals `[-2, 1], [1, 2]`:

```
>>> from scipy.interpolate import PPoly
>>> pp = PPoly(np.array([[1, -4, 3], [1, 0, 0]]).T, [-2, 1, 2])
>>> pp.roots()
array([-1.,  1.])
```

`PPoly.`**`roots`**`(`*discontinuity=True*, *extrapolate=None*`)`
Find real roots of the the piecewise polynomial.

Parameters **discontinuity** : bool, optional
Whether to report sign changes across discontinuities at breakpoints as roots.
extrapolate : {bool, 'periodic', None}, optional
If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns **roots** : ndarray
Roots of the polynomial(s).
If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

See also:

PPoly.solve

`PPoly.`**`extend`**`(`*c*, *x*, *right=None*`)`
Add additional breakpoints and coefficients to the polynomial.

Parameters **c** : ndarray, size (k, m, ...)
Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the *self.x* end points.
x : ndarray, size (m,)
Additional breakpoints. Must be sorted in the same order as *self.x* and either to the right or to the left of the current breakpoints.
right
Deprecated argument. Has no effect.
Deprecated since version 0.19.

classmethod `PPoly.`**`from_spline`**`(`*tck*, *extrapolate=None*`)`
Construct a piecewise polynomial from a spline

Parameters

tck
A spline, as returned by *splrep* or a BSpline object.

extrapolate : bool or 'periodic', optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

classmethod PPoly.**from_bernstein_basis**(*bp*, *extrapolate=None*)
Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

Parameters

bp : BPoly
A Bernstein basis polynomial, as created by BPoly

extrapolate : bool or 'periodic', optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

PPoly.**construct_fast**(*c*, *x*, *extrapolate=None*, *axis=0*)
Construct the piecewise polynomial without making checks.

Takes the same parameters as the constructor. Input arguments `c` and `x` must be arrays of the correct shape and type. The `c` array can only be of dtypes float and complex, and `x` array must have dtype float.

class scipy.interpolate.**BPoly**(*c*, *x*, *extrapolate=None*, *axis=0*)
Piecewise polynomial in terms of coefficients and breakpoints.

The polynomial between `x[i]` and `x[i + 1]` is written in the Bernstein polynomial basis:

```
S = sum(c[a, i] * b(a, k; x) for a in range(k+1)),
```

where `k` is the degree of the polynomial, and:

```
b(a, k; x) = binom(k, a) * t**a * (1 - t)**(k - a),
```

with `t = (x - x[i]) / (x[i+1] - x[i])` and `binom` is the binomial coefficient.

Parameters

c : ndarray, shape (k, m, ...)
Polynomial coefficients, order *k* and *m* intervals

x : ndarray, shape (m+1,)
Polynomial breakpoints. Must be sorted in either increasing or decreasing order.

extrapolate : bool, optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

axis : int, optional
Interpolation axis. Default is zero.

See also:

PPoly piecewise polynomials in the power basis

Notes

Properties of Bernstein polynomials are well documented in the literature. Here's a non-exhaustive list:

Examples

```
>>> from scipy.interpolate import BPoly
>>> x = [0, 1]
>>> c = [[1], [2], [3]]
>>> bp = BPoly(c, x)
```

This creates a 2nd order polynomial

$$B(x) = 1 \times b_{0,2}(x) + 2 \times b_{1,2}(x) + 3 \times b_{2,2}(x)$$
$$= 1 \times (1-x)^2 + 2 \times 2x(1-x) + 3 \times x^2$$

Attributes

x	(ndarray) Breakpoints.
c	(ndarray) Coefficients of the polynomials. They are reshaped to a 3-dimensional array with the last dimension representing the trailing dimensions of the original coefficient array.
axis	(int) Interpolation axis.

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative.
`extend`(c, x[, right])	Add additional breakpoints and coefficients to the polynomial.
`derivative`([nu])	Construct a new piecewise polynomial representing the derivative.
`antiderivative`([nu])	Construct a new piecewise polynomial representing the antiderivative.
`integrate`(a, b[, extrapolate])	Compute a definite integral over a piecewise polynomial.
`construct_fast`(c, x[, extrapolate, axis])	Construct the piecewise polynomial without making checks.
`from_power_basis`(pp[, extrapolate])	Construct a piecewise polynomial in Bernstein basis from a power basis polynomial.
`from_derivatives`(xi, yi[, orders, extrapolate])	Construct a piecewise polynomial in the Bernstein basis, compatible with the specified values and derivatives at breakpoints.

`BPoly.__call__`(*x*, *nu=0*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative.

Parameters

x : array_like
Points to evaluate the interpolant at.

nu : int, optional
Order of derivative to evaluate. Must be non-negative.

extrapolate : {bool, 'periodic', None}, optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns

y : array_like
Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`BPoly.`**`extend`**(*c*, *x*, *right=None*)
Add additional breakpoints and coefficients to the polynomial.

Parameters

c : ndarray, size (k, m, ...)
Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the *self.x* end points.

x : ndarray, size (m,)
Additional breakpoints. Must be sorted in the same order as *self.x* and either to the right or to the left of the current breakpoints.

right
Deprecated argument. Has no effect.
Deprecated since version 0.19.

`BPoly.`**`derivative`**(*nu=1*)
Construct a new piecewise polynomial representing the derivative.

Parameters

nu : int, optional
Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns

bp : BPoly
Piecewise polynomial of order k - nu representing the derivative of this polynomial.

`BPoly.`**`antiderivative`**(*nu=1*)
Construct a new piecewise polynomial representing the antiderivative.

Parameters

nu : int, optional
Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns

bp : BPoly
Piecewise polynomial of order k + nu representing the antiderivative of this polynomial.

Notes

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`BPoly.`**`integrate`**(*a*, *b*, *extrapolate=None*)
Compute a definite integral over a piecewise polynomial.

Parameters

a : float
Lower integration bound

b : float
Upper integration bound

extrapolate : {bool, 'periodic', None}, optional
Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns

array_like
Definite integral of the piecewise polynomial over [a, b]

BPoly.**construct_fast**(*c*, *x*, *extrapolate=None*, *axis=0*)
Construct the piecewise polynomial without making checks.

Takes the same parameters as the constructor. Input arguments `c` and `x` must be arrays of the correct shape and type. The `c` array can only be of dtypes float and complex, and `x` array must have dtype float.

classmethod BPoly.**from_power_basis**(*pp*, *extrapolate=None*)
Construct a piecewise polynomial in Bernstein basis from a power basis polynomial.

Parameters **pp** : PPoly
A piecewise polynomial in the power basis
extrapolate : bool or 'periodic', optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

classmethod BPoly.**from_derivatives**(*xi*, *yi*, *orders=None*, *extrapolate=None*)
Construct a piecewise polynomial in the Bernstein basis, compatible with the specified values and derivatives at breakpoints.

Parameters **xi** : array_like
sorted 1D array of x-coordinates
yi : array_like or list of array_likes
`yi[i][j]` is the `j`-th derivative known at `xi[i]`
orders : None or int or array_like of ints. Default: None.
Specifies the degree of local polynomials. If not None, some derivatives are ignored.
extrapolate : bool or 'periodic', optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

Notes

If `k` derivatives are specified at a breakpoint `x`, the constructed polynomial is exactly `k` times continuously differentiable at `x`, unless the `order` is provided explicitly. In the latter case, the smoothness of the polynomial at the breakpoint is controlled by the `order`.

Deduces the number of derivatives to match at each end from `order` and the number of derivatives available. If possible it uses the same number of derivatives from each end; if the number is odd it tries to take the extra one from y2. In any case if not enough derivatives are available at one end or another it draws enough to make up the total from the other end.

If the order is too high and not enough derivatives are available, an exception is raised.

Examples

```
>>> from scipy.interpolate import BPoly
>>> BPoly.from_derivatives([0, 1], [[1, 2], [3, 4]])
```

Creates a polynomial *f(x)* of degree 3, defined on *[0, 1]* such that *f(0) = 1, df/dx(0) = 2, f(1) = 3, df/dx(1) = 4*

```
>>> BPoly.from_derivatives([0, 1, 2], [[0, 1], [0], [2]])
```

Creates a piecewise polynomial *f(x)*, such that *f(0) = f(1) = 0*, *f(2) = 2*, and *df/dx(0) = 1*. Based on the number of derivatives provided, the order of the local polynomials is 2 on *[0, 1]* and 1 on *[1, 2]*. Notice that no restriction is imposed on the derivatives at *x = 1* and *x = 2*.

Indeed, the explicit form of the polynomial is:

```
f(x) = | x * (1 - x),  0 <= x < 1
       | 2 * (x - 1),  1 <= x <= 2
```

So that f'(1-0) = -1 and f'(1+0) = 2

5.7.2 Multivariate interpolation

Unstructured data:

griddata(points, values, xi[, method, ...])	Interpolate unstructured D-dimensional data.
LinearNDInterpolator(points, values[, ...])	Piecewise linear interpolant in N dimensions.
NearestNDInterpolator(points, values)	Nearest-neighbour interpolation in N dimensions.
CloughTocher2DInterpolator(points, values[, tol])	Piecewise cubic, C1 smooth, curvature-minimizing interpolant in 2D.
Rbf(*args)	A class for radial basis function approximation/interpolation of n-dimensional scattered data.
interp2d(x, y, z[, kind, copy, ...])	Interpolate over a 2-D grid.

scipy.interpolate.**griddata**(*points*, *values*, *xi*, *method='linear'*, *fill_value=nan*, *rescale=False*)
Interpolate unstructured D-dimensional data.

Parameters

points : ndarray of floats, shape (n, D)
Data point coordinates. Can either be an array of shape (n, D), or a tuple of *ndim* arrays.

values : ndarray of float or complex, shape (n,)
Data values.

xi : 2-D ndarray of float or tuple of 1-D array, shape (M, D)
Points at which to interpolate data.

method : {'linear', 'nearest', 'cubic'}, optional
Method of interpolation. One of

- ***nearest*** return the value at the data point closest to the point of interpolation. See *NearestNDInterpolator* for more details.
- ***linear*** tesselate the input point set to n-dimensional simplices, and interpolate linearly on each simplex. See *LinearNDInterpolator* for more details.
- ***cubic (1-D)*** return the value determined from a cubic spline.
- ***cubic (2-D)*** return the value determined from a piecewise cubic, continuously differentiable (C1), and approximately curvature-minimizing polynomial surface. See *CloughTocher2DInterpolator* for more details.

fill_value : float, optional
Value used to fill in for requested points outside of the convex hull of the input points. If not provided, then the default is nan. This option has no effect for the 'nearest' method.

rescale : bool, optional
Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.
New in version 0.14.0.

Notes

New in version 0.9.

Examples

Suppose we want to interpolate the 2-D function

```
>>> def func(x, y):
...     return x*(1-x)*np.cos(4*np.pi*x) * np.sin(4*np.pi*y**2)**2
```

on a grid in [0, 1]x[0, 1]

```
>>> grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]
```

but we only know its values at 1000 data points:

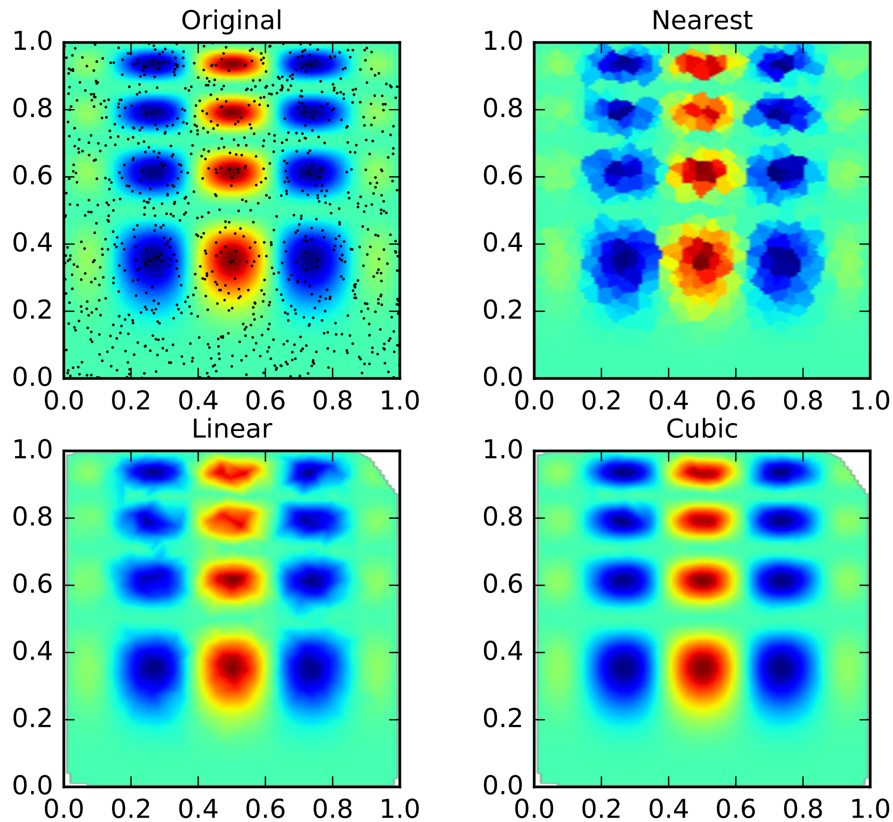
```
>>> points = np.random.rand(1000, 2)
>>> values = func(points[:,0], points[:,1])
```

This can be done with `griddata` – below we try out all of the interpolation methods:

```
>>> from scipy.interpolate import griddata
>>> grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
>>> grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
>>> grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
```

One can see that the exact result is reproduced by all of the methods to some degree, but for this smooth function the piecewise cubic interpolant gives the best results:

```
>>> import matplotlib.pyplot as plt
>>> plt.subplot(221)
>>> plt.imshow(func(grid_x, grid_y).T, extent=(0,1,0,1), origin='lower')
>>> plt.plot(points[:,0], points[:,1], 'k.', ms=1)
>>> plt.title('Original')
>>> plt.subplot(222)
>>> plt.imshow(grid_z0.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Nearest')
>>> plt.subplot(223)
>>> plt.imshow(grid_z1.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Linear')
>>> plt.subplot(224)
>>> plt.imshow(grid_z2.T, extent=(0,1,0,1), origin='lower')
>>> plt.title('Cubic')
>>> plt.gcf().set_size_inches(6, 6)
>>> plt.show()
```



class `scipy.interpolate.`**`LinearNDInterpolator`**(*points*, *values*, *fill_value=np.nan*, *rescale=False*)

Piecewise linear interpolant in N dimensions.

New in version 0.9.

Parameters

points : ndarray of floats, shape (npoints, ndims); or Delaunay
  Data point coordinates, or a precomputed Delaunay triangulation.

values : ndarray of float or complex, shape (npoints, ...)
  Data values.

fill_value : float, optional
  Value used to fill in for requested points outside of the convex hull of the input points. If not provided, then the default is `nan`.

rescale : bool, optional
  Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

Notes

The interpolant is constructed by triangulating the input data with Qhull *[R61]*, and on each triangle performing linear barycentric interpolation.

References

[R61]

Methods

`__call__`(xi)	Evaluate interpolator at given points.

`LinearNDInterpolator.__call__(xi)`
Evaluate interpolator at given points.

Parameters **xi** : ndarray of float, shape (..., ndim)
Points where to interpolate data at.

class `scipy.interpolate.NearestNDInterpolator(points, values)`
Nearest-neighbour interpolation in N dimensions.

New in version 0.9.

Parameters

x : (Npoints, Ndims) ndarray of floats
Data point coordinates.

y : (Npoints,) ndarray of float or complex
Data values.

rescale : boolean, optional
Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.
New in version 0.14.0.

tree_options : dict, optional
Options passed to the underlying `cKDTree`.
New in version 0.17.0.

Notes

Uses `scipy.spatial.cKDTree`

Methods

`__call__`(*args)	Evaluate interpolator at given points.

`NearestNDInterpolator.__call__(*args)`
Evaluate interpolator at given points.

Parameters **xi** : ndarray of float, shape (..., ndim)
Points where to interpolate data at.

class `scipy.interpolate.CloughTocher2DInterpolator(points, values, tol=1e-6)`
Piecewise cubic, C1 smooth, curvature-minimizing interpolant in 2D.

New in version 0.9.

Parameters **points** : ndarray of floats, shape (npoints, ndims); or Delaunay
Data point coordinates, or a precomputed Delaunay triangulation.

values : ndarray of float or complex, shape (npoints, ...)
Data values.
fill_value : float, optional
Value used to fill in for requested points outside of the convex hull of the input points. If not provided, then the default is `nan`.
tol : float, optional
Absolute/relative tolerance for gradient estimation.
maxiter : int, optional
Maximum number of iterations in gradient estimation.
rescale : bool, optional
Rescale points to unit cube before performing interpolation. This is useful if some of the input dimensions have incommensurable units and differ by many orders of magnitude.

Notes

The interpolant is constructed by triangulating the input data with Qhull *[R57]*, and constructing a piecewise cubic interpolating Bezier polynomial on each triangle, using a Clough-Tocher scheme *[CT]*. The interpolant is guaranteed to be continuously differentiable.

The gradients of the interpolant are chosen so that the curvature of the interpolating surface is approximatively minimized. The gradients necessary for this are estimated using the global algorithm described in [Nielson83,Renka84]_.

References

[R57], *[CT]*, *[Nielson83]*, *[Renka84]*

Methods

`__call__`(xi)	Evaluate interpolator at given points.

`CloughTocher2DInterpolator.__call__(xi)`
Evaluate interpolator at given points.

Parameters **xi** : ndarray of float, shape (..., ndim)
Points where to interpolate data at.

class `scipy.interpolate.Rbf(*args)`
A class for radial basis function approximation/interpolation of n-dimensional scattered data.

Parameters ***args** : arrays
x, y, z, ..., d, where x, y, z, ... are the coordinates of the nodes and d is the array of values at the nodes
function : str or callable, optional
The radial basis function, based on the radius, r, given by the norm (default is Euclidean distance); the default is 'multiquadric':

```
'multiquadric': sqrt((r/self.epsilon)**2 + 1)
'inverse': 1.0/sqrt((r/self.epsilon)**2 + 1)
'gaussian': exp(-(r/self.epsilon)**2)
'linear': r
'cubic': r**3
'quintic': r**5
'thin_plate': r**2 * log(r)
```

If callable, then it must take 2 arguments (self, r). The epsilon parameter will be available as self.epsilon. Other keyword arguments passed in will be available as well.

epsilon : float, optional
Adjustable constant for gaussian or multiquadrics functions - defaults to approximate average distance between nodes (which is a good start).

smooth : float, optional
Values greater than zero increase the smoothness of the approximation. 0 is for interpolation (default), the function will always go through the nodal points in this case.

norm : callable, optional
A function that returns the 'distance' between two points, with inputs as arrays of positions (x, y, z, ...), and an output as an array of distance. E.g, the default:

```
def euclidean_norm(x1, x2):
    return sqrt( ((x1 - x2)**2).sum(axis=0) )
```

which is called with x1=x1[ndims,newaxis,:] and x2=x2[ndims,:,newaxis] such that the result is a matrix of the distances from each point in x1 to each point in x2.

Examples

```
>>> from scipy.interpolate import Rbf
>>> x, y, z, d = np.random.rand(4, 50)
>>> rbfi = Rbf(x, y, z, d)  # radial basis function interpolator instance
>>> xi = yi = zi = np.linspace(0, 1, 20)
>>> di = rbfi(xi, yi, zi)   # interpolated values
>>> di.shape
(20,)
```

Methods

`__call__`(*args)

`Rbf.__call__`(*args)

class `scipy.interpolate.interp2d`(*x*, *y*, *z*, *kind='linear'*, *copy=True*, *bounds_error=False*, *fill_value=nan*)

Interpolate over a 2-D grid.

x, *y* and *z* are arrays of values used to approximate some function f: `z = f(x, y)`. This class returns a function whose call method uses spline interpolation to find the value of new points.

If *x* and *y* represent a regular grid, consider using RectBivariateSpline.

Note that calling *interp2d* with NaNs present in input values results in undefined behaviour.

Parameters

x, y : array_like
Arrays defining the data point coordinates.
If the points lie on a regular grid, *x* can specify the column coordinates and *y* the row coordinates, for example:

```
>>> x = [0,1,2];  y = [0,3]; z = [[1,2,3], [4,5,6]]
```

Otherwise, *x* and *y* must specify the full coordinates for each point, for example:

```
>>> x = [0,1,2,0,1,2];  y = [0,0,0,3,3,3]; z = [1,2,3,4,5,6]
```

If *x* and *y* are multi-dimensional, they are flattened before use.

z : array_like
The values of the function to interpolate at the data points. If *z* is a multi-dimensional array, it is flattened before use. The length of a flattened *z* array is either len(*x*)*len(*y*)

if *x* and *y* specify the column and row coordinates or `len(z) == len(x) == len(y)` if *x* and *y* specify coordinates for each point.

kind : {'linear', 'cubic', 'quintic'}, optional
The kind of spline interpolation to use. Default is 'linear'.

copy : bool, optional
If True, the class makes internal copies of x, y and z. If False, references may be used. The default is to copy.

bounds_error : bool, optional
If True, when interpolated values are requested outside of the domain of the input data (x,y), a ValueError is raised. If False, then *fill_value* is used.

fill_value : number, optional
If provided, the value to use for points outside of the interpolation domain. If omitted (None), values outside the domain are extrapolated.

See also:

`RectBivariateSpline` Much faster 2D interpolation if your input data is on a grid

`bisplrep`, *`bisplev`*

`BivariateSpline` a more recent wrapper of the FITPACK routines

`interp1d` one dimension version of this function

Notes

The minimum number of data points required along the interpolation axis is `(k+1)**2`, with k=1 for linear, k=3 for cubic and k=5 for quintic interpolation.

The interpolator is constructed by *`bisplrep`*, with a smoothing factor of 0. If more control over smoothing is needed, *`bisplrep`* should be used directly.

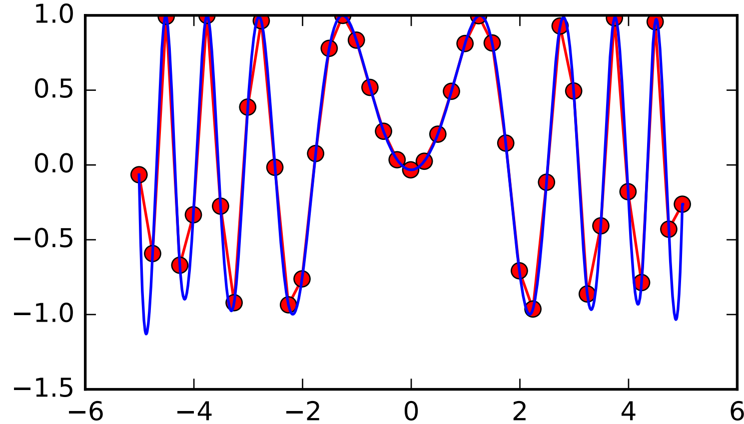
Examples

Construct a 2-D grid and interpolate on it:

```
>>> from scipy import interpolate
>>> x = np.arange(-5.01, 5.01, 0.25)
>>> y = np.arange(-5.01, 5.01, 0.25)
>>> xx, yy = np.meshgrid(x, y)
>>> z = np.sin(xx**2+yy**2)
>>> f = interpolate.interp2d(x, y, z, kind='cubic')
```

Now use the obtained interpolation function and plot the result:

```
>>> import matplotlib.pyplot as plt
>>> xnew = np.arange(-5.01, 5.01, 1e-2)
>>> ynew = np.arange(-5.01, 5.01, 1e-2)
>>> znew = f(xnew, ynew)
>>> plt.plot(x, z[0, :], 'ro-', xnew, znew[0, :], 'b-')
>>> plt.show()
```



Methods

`__call__`(x, y[, dx, dy, assume_sorted])	Interpolate the function.

`interp2d.__call__`(*x*, *y*, *dx=0*, *dy=0*, *assume_sorted=False*)

Interpolate the function.

Parameters

x : 1D array
    x-coordinates of the mesh on which to interpolate.

y : 1D array
    y-coordinates of the mesh on which to interpolate.

dx : int >= 0, < kx
    Order of partial derivatives in x.

dy : int >= 0, < ky
    Order of partial derivatives in y.

assume_sorted : bool, optional
    If False, values of *x* and *y* can be in any order and they are sorted first. If True, *x* and *y* have to be arrays of monotonically increasing values.

Returns

z : 2D array with shape (len(y), len(x))
    The interpolated values.

For data on a grid:

`interpn`(points, values, xi[, method, ...])	Multidimensional interpolation on regular grids.
`RegularGridInterpolator`(points, values[, ...])	Interpolation on a regular grid in arbitrary dimensions
`RectBivariateSpline`(x, y, z[, bbox, kx, ky, s])	Bivariate spline approximation over a rectangular mesh.

`scipy.interpolate.interpn`(*points*, *values*, *xi*, *method='linear'*, *bounds_error=True*, *fill_value=nan*)

Multidimensional interpolation on regular grids.

Parameters

points : tuple of ndarray of float, with shapes (m1,), ..., (mn,)
    The points defining the regular grid in n dimensions.

values : array_like, shape (m1, ..., mn, ...)
    The data on the regular grid in n dimensions.

xi : ndarray of shape (..., ndim)
The coordinates to sample the gridded data at

method : str, optional
The method of interpolation to perform. Supported are "linear" and "nearest", and "splinef2d". "splinef2d" is only supported for 2-dimensional data.

bounds_error : bool, optional
If True, when interpolated values are requested outside of the domain of the input data, a ValueError is raised. If False, then *fill_value* is used.

fill_value : number, optional
If provided, the value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated. Extrapolation is not supported by method "splinef2d".

Returns **values_x** : ndarray, shape xi.shape[:-1] + values.shape[ndim:]
Interpolated values at input coordinates.

See also:

NearestNDInterpolator
Nearest neighbour interpolation on unstructured data in N dimensions

LinearNDInterpolator
Piecewise linear interpolant on unstructured data in N dimensions

RegularGridInterpolator
Linear and nearest-neighbor Interpolation on a regular grid in arbitrary dimensions

RectBivariateSpline
Bivariate spline approximation over a rectangular mesh

Notes

New in version 0.14.

class scipy.interpolate.**RegularGridInterpolator**(*points*, *values*, *method='linear'*, *bounds_error=True*, *fill_value=nan*)

Interpolation on a regular grid in arbitrary dimensions

The data must be defined on a regular grid; the grid spacing however may be uneven. Linear and nearest-neighbour interpolation are supported. After setting up the interpolator object, the interpolation method (*linear* or *nearest*) may be chosen at each evaluation.

Parameters **points** : tuple of ndarray of float, with shapes (m1,), ..., (mn,)
The points defining the regular grid in n dimensions.

values : array_like, shape (m1, ..., mn, ...)
The data on the regular grid in n dimensions.

method : str, optional
The method of interpolation to perform. Supported are "linear" and "nearest". This parameter will become the default for the object's `__call__` method. Default is "linear".

bounds_error : bool, optional
If True, when interpolated values are requested outside of the domain of the input data, a ValueError is raised. If False, then *fill_value* is used.

fill_value : number, optional
If provided, the value to use for points outside of the interpolation domain. If None, values outside the domain are extrapolated.

See also:

NearestNDInterpolator
Nearest neighbour interpolation on unstructured data in N dimensions

LinearNDInterpolator
Piecewise linear interpolant on unstructured data in N dimensions

Notes

Contrary to LinearNDInterpolator and NearestNDInterpolator, this class avoids expensive triangulation of the input data by taking advantage of the regular grid structure.

New in version 0.14.

References

[R64], *[R65]*, *[R66]*

Examples

Evaluate a simple example function on the points of a 3D grid:

```
>>> from scipy.interpolate import RegularGridInterpolator
>>> def f(x, y, z):
...     return 2 * x**3 + 3 * y**2 - z
>>> x = np.linspace(1, 4, 11)
>>> y = np.linspace(4, 7, 22)
>>> z = np.linspace(7, 9, 33)
>>> data = f(*np.meshgrid(x, y, z, indexing='ij', sparse=True))
```

`data` is now a 3D array with `data[i,j,k] = f(x[i], y[j], z[k])`. Next, define an interpolating function from this data:

```
>>> my_interpolating_function = RegularGridInterpolator((x, y, z), data)
```

Evaluate the interpolating function at the two points `(x,y,z) = (2.1, 6.2, 8.3)` and `(3.3, 5.2, 7.1)`:

```
>>> pts = np.array([[2.1, 6.2, 8.3], [3.3, 5.2, 7.1]])
>>> my_interpolating_function(pts)
array([ 125.80469388,  146.30069388])
```

which is indeed a close approximation to `[f(2.1, 6.2, 8.3), f(3.3, 5.2, 7.1)]`.

Methods

`__call__`(xi[, method])	Interpolation at coordinates

`RegularGridInterpolator.`**`__call__`**(*xi, method=None*)
Interpolation at coordinates

Parameters
xi : ndarray of shape (..., ndim)
The coordinates to sample the gridded data at
method : str
The method of interpolation to perform. Supported are "linear" and "nearest".

class `scipy.interpolate.`**`RectBivariateSpline`**(*x, y, z, bbox=[None, None, None, None], kx=3, ky=3, s=0*)
Bivariate spline approximation over a rectangular mesh.

Can be used for both smoothing and interpolating data.

Parameters **x,y** : array_like
  1-D arrays of coordinates in strictly ascending order.
z : array_like
  2-D array of data with shape (x.size,y.size).
bbox : array_like, optional
  Sequence of length 4 specifying the boundary of the rectangular approximation domain. By default, `bbox=[min(x,tx),max(x,tx), min(y,ty),max(y,ty)]`.
kx, ky : ints, optional
  Degrees of the bivariate spline. Default is 3.
s : float, optional
  Positive smoothing factor defined for estimation condition: `sum((w[i]*(z[i]-s(x[i], y[i])))**2, axis=0) <= s` Default is `s=0`, which is for interpolation.

See also:

`SmoothBivariateSpline` a smoothing bivariate spline for scattered data

`bisplrep` an older wrapping of FITPACK

`bisplev` an older wrapping of FITPACK

`UnivariateSpline` a similar class for univariate spline interpolation

Methods

`__call__`(x, y[, mth, dx, dy, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(xi, yi[, dx, dy])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.
`get_knots`()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
`get_residual`()	Return weighted sum of squared residuals of the spline
`integral`(xa, xb, ya, yb)	Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

`RectBivariateSpline.`**`__call__`**(*x*, *y*, *mth=None*, *dx=0*, *dy=0*, *grid=True*)
  Evaluate the spline or its derivatives at given positions.

Parameters **x, y** : array_like
  Input coordinates.
  If *grid* is False, evaluate the spline at points `(x[i], y[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.
  If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays x, y. The arrays must be sorted to increasing order.
dx : int
  Order of x-derivative
  New in version 0.14.0.
dy : int
  Order of y-derivative
  New in version 0.14.0.
grid : bool

Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.
New in version 0.14.0.

mth : str
Deprecated argument. Has no effect.

`RectBivariateSpline.`**`ev`**`(`*xi*, *yi*, *dx=0*, *dy=0*`)`
Evaluate the spline at points

Returns the interpolated value at `(xi[i], yi[i]), i=0,...,len(xi)-1`.

Parameters **xi, yi** : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.
dx : int, optional
Order of x-derivative
New in version 0.14.0.
dy : int, optional
Order of y-derivative
New in version 0.14.0.

`RectBivariateSpline.`**`get_coeffs`**`()`
Return spline coefficients.

`RectBivariateSpline.`**`get_knots`**`()`
Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`RectBivariateSpline.`**`get_residual`**`()`
Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

`RectBivariateSpline.`**`integral`**`(`*xa*, *xb*, *ya*, *yb*`)`
Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

Parameters **xa, xb** : float
The end-points of the x integration interval.
ya, yb : float
The end-points of the y integration interval.

Returns **integ** : float
The value of the resulting integral.

See also:

`scipy.ndimage.map_coordinates`

Tensor product polynomials:

`NdPPoly`(c, x[, extrapolate])	Piecewise tensor product polynomial

class `scipy.interpolate.`**`NdPPoly`**`(`*c*, *x*, *extrapolate=None*`)`
Piecewise tensor product polynomial

The value at point *xp = (x', y', z', ...)* is evaluated by first computing the interval indices *i* such that:

```
x[0][i[0]] <= x' < x[0][i[0]+1]
x[1][i[1]] <= y' < x[1][i[1]+1]
...
```

and then computing:

```
S = sum(c[k0-m0-1,...,kn-mn-1,i[0],...,i[n]]
        * (xp[0] - x[0][i[0]])**m0
        * ...
        * (xp[n] - x[n][i[n]])**mn
        for m0 in range(k[0]+1)
        ...
        for mn in range(k[n]+1))
```

where `k[j]` is the degree of the polynomial in dimension j. This representation is the piecewise multivariate power basis.

Parameters

c : ndarray, shape (k0, ..., kn, m0, ..., mn, ...)
: Polynomial coefficients, with polynomial order *kj* and *mj+1* intervals for each dimension *j*.

x : ndim-tuple of ndarrays, shapes (mj+1,)
: Polynomial breakpoints for each dimension. These must be sorted in increasing order.

extrapolate : bool, optional
: Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. Default: True.

See also:

`PPoly` piecewise polynomials in 1D

Notes

High-order polynomials in the power basis can be numerically unstable.

Attributes

x	(tuple of ndarrays) Breakpoints.
c	(ndarray) Coefficients of the polynomials.

Methods

`__call__`(x[, nu, extrapolate])	Evaluate the piecewise polynomial or its derivative
`construct_fast`(c, x[, extrapolate])	Construct the piecewise polynomial without making checks.

`NdPPoly.`**`__call__`**(*x*, *nu=None*, *extrapolate=None*)

Evaluate the piecewise polynomial or its derivative

Parameters

x : array-like
: Points to evaluate the interpolant at.

nu : tuple, optional
: Orders of derivatives to evaluate. Each must be non-negative.

extrapolate : bool, optional
: Whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs.

Returns

y : array-like
: Interpolated values. Shape is determined by replacing the interpolation axis in the original array with the shape of x.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

classmethod `NdPPoly.`**`construct_fast`**(*c*, *x*, *extrapolate=None*)
Construct the piecewise polynomial without making checks.

Takes the same parameters as the constructor. Input arguments *c* and *x* must be arrays of the correct shape and type. The *c* array can only be of dtypes float and complex, and *x* array must have dtype float.

5.7.3 1-D Splines

BSpline(t, c, k[, extrapolate, axis])	Univariate spline in the B-spline basis.
make_interp_spline(x, y[, k, t, bc_type, ...])	Compute the (coefficients of) interpolating B-spline.
make_lsq_spline(x, y, t[, k, w, axis, ...])	Compute the (coefficients of) an LSQ B-spline.

class `scipy.interpolate.`**`BSpline`**(*t*, *c*, *k*, *extrapolate=True*, *axis=0*)
Univariate spline in the B-spline basis.

$$S(x) = \sum_{j=0}^{n-1} c_j B_{j,k;t}(x)$$

where $B_{j,k;t}$ are B-spline basis functions of degree *k* and knots *t*.

Parameters

t : ndarray, shape (n+k+1,)
knots

c : ndarray, shape (>=n, ...)
spline coefficients

k : int
B-spline order

extrapolate : bool, optional
whether to extrapolate beyond the base interval, `t[k] .. t[n]`, or to return nans. If True, extrapolates the first and last polynomial pieces of b-spline functions active on the base interval. Default is True.

axis : int, optional
Interpolation axis. Default is zero.

Notes

B-spline basis elements are defined via

$$B_{i,0}(x) = 1, \text{if } t_i \le x < t_{i+1}, \text{otherwise } 0,$$

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x)$$

Implementation details

- At least `k+1` coefficients are required for a spline of degree *k*, so that `n >= k+1`. Additional coefficients, `c[j]` with `j > n`, are ignored.
- B-spline basis elements of degree *k* form a partition of unity on the *base interval*, `t[k] <= x <= t[n]`.

References

[R55], [R56]

Examples

Translating the recursive definition of B-splines into Python code, we have:

```
>>> def B(x, k, i, t):
...     if k == 0:
...        return 1.0 if t[i] <= x < t[i+1] else 0.0
...     if t[i+k] == t[i]:
...        c1 = 0.0
...     else:
...        c1 = (x - t[i])/(t[i+k] - t[i]) * B(x, k-1, i, t)
...     if t[i+k+1] == t[i+1]:
...        c2 = 0.0
...     else:
...        c2 = (t[i+k+1] - x)/(t[i+k+1] - t[i+1]) * B(x, k-1, i+1, t)
...     return c1 + c2
```

```
>>> def bspline(x, t, c, k):
...     n = len(t) - k - 1
...     assert (n >= k+1) and (len(c) >= n)
...     return sum(c[i] * B(x, k, i, t) for i in range(n))
```

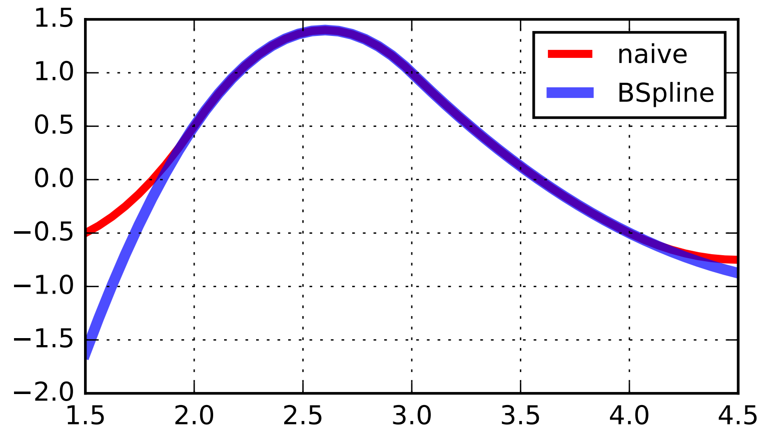
Note that this is an inefficient (if straightforward) way to evaluate B-splines — this spline class does it in an equivalent, but much more efficient way.

Here we construct a quadratic spline function on the base interval `2 <= x <= 4` and compare with the naive way of evaluating the spline:

```
>>> from scipy.interpolate import BSpline
>>> k = 2
>>> t = [0, 1, 2, 3, 4, 5, 6]
>>> c = [-1, 2, 0, -1]
>>> spl = BSpline(t, c, k)
>>> spl(2.5)
array(1.375)
>>> bspline(2.5, t, c, k)
1.375
```

Note that outside of the base interval results differ. This is because *BSpline* extrapolates the first and last polynomial pieces of b-spline functions active on the base interval.

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> xx = np.linspace(1.5, 4.5, 50)
>>> ax.plot(xx, [bspline(x, t, c ,k) for x in xx], 'r-', lw=3, label='naive')
>>> ax.plot(xx, spl(xx), 'b-', lw=4, alpha=0.7, label='BSpline')
>>> ax.grid(True)
>>> ax.legend(loc='best')
>>> plt.show()
```



Attributes

tck	Equivalent to `(self.t, self.c, self.k)` (read-only).

`BSpline.`**`tck`**

Equivalent to `(self.t, self.c, self.k)` (read-only).

t	(ndarray) knot vector
c	(ndarray) spline coefficients
k	(int) spline degree
extrapolate	(bool) If True, extrapolates the first and last polynomial pieces of b-spline functions active on the base interval.
axis	(int) Interpolation axis.

Methods

__call__(x[, nu, extrapolate])	Evaluate a spline function.
basis_element(t[, extrapolate])	Return a B-spline basis element `B(x | t[0], ..., t[k+1])`.
derivative([nu])	Return a b-spline representing the derivative.
antiderivative([nu])	Return a b-spline representing the antiderivative.
integrate(a, b[, extrapolate])	Compute a definite integral of the spline.
construct_fast(t, c, k[, extrapolate, axis])	Construct a spline without making checks.

`BSpline.`**`__call__`**(*x*, *nu=0*, *extrapolate=None*)

Evaluate a spline function.

Parameters **x** : array_like

points to evaluate the spline at.

nu: int, optional

derivative to evaluate (default is 0).

extrapolate : bool, optional

whether to extrapolate based on the first and last intervals or return nans. Default is *self.extrapolate*.

Returns **y** : array_like
Shape is determined by replacing the interpolation axis in the coefficient array with the shape of *x*.

classmethod `BSpline.basis_element` (*t*, *extrapolate=True*)
Return a B-spline basis element `B(x | t[0], ..., t[k+1])`.

Parameters **t** : ndarray, shape (k+1,)
internal knots

extrapolate : bool, optional
whether to extrapolate beyond the base interval, `t[0] .. t[k+1]`, or to return nans. Default is True.

Returns **basis_element** : callable
A callable representing a B-spline basis element for the knot vector *t*.

Notes

The order of the b-spline, *k*, is inferred from the length of *t* as `len(t)-2`. The knot vector is constructed by appending and prepending `k+1` elements to internal knots *t*.

Examples

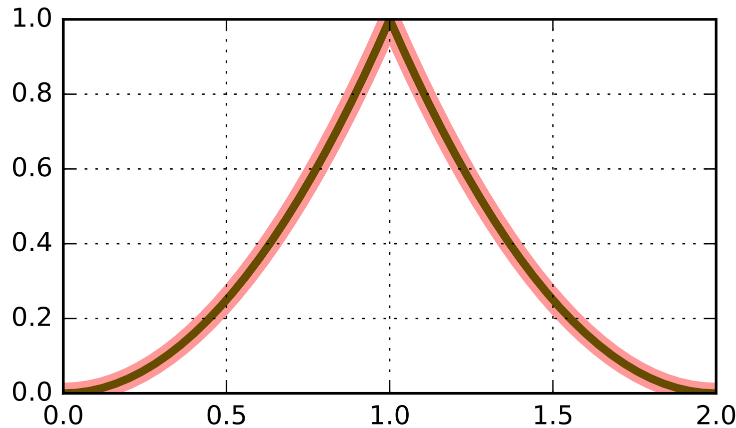
Construct a cubic b-spline:

```
>>> from scipy.interpolate import BSpline
>>> b = BSpline.basis_element([0, 1, 2, 3, 4])
>>> k = b.k
>>> b.t[k:-k]
array([ 0.,  1.,  2.,  3.,  4.])
>>> k
3
```

Construct a second order b-spline on `[0, 1, 1, 2]`, and compare to its explicit form:

```
>>> t = [-1, 0, 1, 1, 2]
>>> b = BSpline.basis_element(t[1:])
>>> def f(x):
...     return np.where(x < 1, x*x, (2. - x)**2)
```

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> x = np.linspace(0, 2, 51)
>>> ax.plot(x, b(x), 'g', lw=3)
>>> ax.plot(x, f(x), 'r', lw=8, alpha=0.4)
>>> ax.grid(True)
>>> plt.show()
```



`BSpline.`**`derivative`**`(`*nu=1*`)`

Return a b-spline representing the derivative.

Parameters **nu** : int, optional
Derivative order. Default is 1.

Returns **b** : BSpline object
A new instance representing the derivative.

See also:

`splder`, *`splantider`*

`BSpline.`**`antiderivative`**`(`*nu=1*`)`

Return a b-spline representing the antiderivative.

Parameters **nu** : int, optional
Antiderivative order. Default is 1.

Returns **b** : BSpline object
A new instance representing the antiderivative.

See also:

`splder`, *`splantider`*

`BSpline.`**`integrate`**`(`*a*, *b*, *extrapolate=None*`)`

Compute a definite integral of the spline.

Parameters **a** : float
Lower limit of integration.

b : float
Upper limit of integration.

extrapolate : bool, optional
whether to extrapolate beyond the base interval, `t[k] .. t[-k-1]`, or take the spline to be zero outside of the base interval. Default is True.

Returns **I** : array_like
Definite integral of the spline over the interval `[a, b]`.

Examples

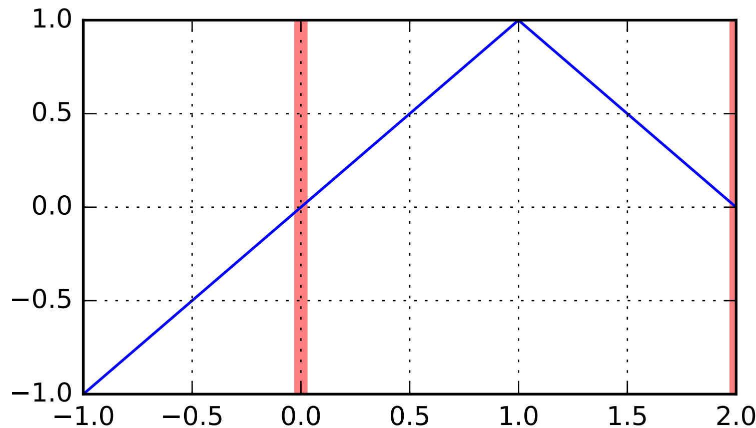
Construct the linear spline `x if x < 1 else 2 - x` on the base interval $[0, 2]$, and integrate it

```
>>> from scipy.interpolate import BSpline
>>> b = BSpline.basis_element([0, 1, 2])
>>> b.integrate(0, 1)
array(0.5)
```

If the integration limits are outside of the base interval, the result is controlled by the *extrapolate* parameter

```
>>> b.integrate(-1, 1)
array(0.0)
>>> b.integrate(-1, 1, extrapolate=False)
array(0.5)
```

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> ax.grid(True)
>>> ax.axvline(0, c='r', lw=5, alpha=0.5)  # base interval
>>> ax.axvline(2, c='r', lw=5, alpha=0.5)
>>> xx = [-1, 1, 2]
>>> ax.plot(xx, b(xx))
>>> plt.show()
```



classmethod `BSpline.`**`construct_fast`**(*t*, *c*, *k*, *extrapolate=True*, *axis=0*)

Construct a spline without making checks.

Accepts same parameters as the regular constructor. Input arrays *t* and *c* must of correct shape and dtype.

`scipy.interpolate.`**`make_interp_spline`**(*x*, *y*, *k=3*, *t=None*, *bc_type=None*, *axis=0*, *check_finite=True*)

Compute the (coefficients of) interpolating B-spline.

Parameters **x** : array_like, shape (n,)

Abscissas.

y : array_like, shape (n, ...)

Ordinates.

k : int, optional

B-spline degree. Default is cubic, k=3.

t : array_like, shape (nt + k + 1,), optional.
Knots. The number of knots needs to agree with the number of datapoints and the number of derivatives at the edges. Specifically, `nt - n` must equal `len(deriv_l) + len(deriv_r)`.

bc_type : 2-tuple or None
Boundary conditions. Default is None, which means choosing the boundary conditions automatically. Otherwise, it must be a length-two tuple where the first element sets the boundary conditions at `x[0]` and the second element sets the boundary conditions at `x[-1]`. Each of these must be an iterable of pairs `(order, value)` which gives the values of derivatives of specified orders at the given edge of the interpolation interval.

axis : int, optional
Interpolation axis. Default is 0.

check_finite : bool, optional
Whether to check that the input arrays contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is True.

Returns **b** : a BSpline object of the degree `k` and with knots `t`.

See also:

`BSpline` base class representing the B-spline objects

`CubicSpline` a cubic spline in the polynomial basis

`make_lsq_spline` a similar factory function for spline fitting

`UnivariateSpline` a wrapper over FITPACK spline fitting routines

`splrep` a wrapper over FITPACK spline fitting routines

Examples

Use cubic interpolation on Chebyshev nodes:

```
>>> def cheb_nodes(N):
...     jj = 2.*np.arange(N) + 1
...     x = np.cos(np.pi * jj / 2 / N)[::-1]
...     return x
```

```
>>> x = cheb_nodes(20)
>>> y = np.sqrt(1 - x**2)
```

```
>>> from scipy.interpolate import BSpline, make_interp_spline
>>> b = make_interp_spline(x, y)
>>> np.allclose(b(x), y)
True
```

Note that the default is a cubic spline with a not-a-knot boundary condition

```
>>> b.k
3
```

Here we use a 'natural' spline, with zero 2nd derivatives at edges:

```
>>> l, r = [(2, 0)], [(2, 0)]
>>> b_n = make_interp_spline(x, y, bc_type=(l, r))
>>> np.allclose(b_n(x), y)
True
>>> x0, x1 = x[0], x[-1]
>>> np.allclose([b_n(x0, 2), b_n(x1, 2)], [0, 0])
True
```

Interpolation of parametric curves is also supported. As an example, we compute a discretization of a snail curve in polar coordinates

```
>>> phi = np.linspace(0, 2.*np.pi, 40)
>>> r = 0.3 + np.cos(phi)
>>> x, y = r*np.cos(phi), r*np.sin(phi)  # convert to Cartesian coordinates
```

Build an interpolating curve, parameterizing it by the angle

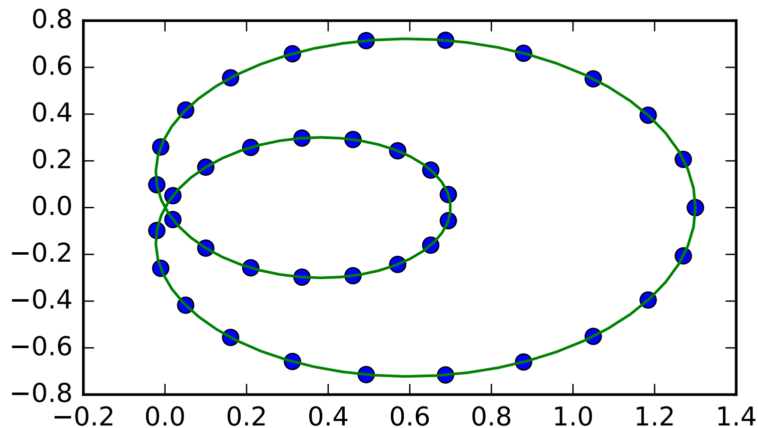
```
>>> from scipy.interpolate import make_interp_spline
>>> spl = make_interp_spline(phi, np.c_[x, y])
```

Evaluate the interpolant on a finer grid (note that we transpose the result to unpack it into a pair of x- and y-arrays)

```
>>> phi_new = np.linspace(0, 2.*np.pi, 100)
>>> x_new, y_new = spl(phi_new).T
```

Plot the result

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
>>> plt.plot(x_new, y_new, '-')
>>> plt.show()
```



`scipy.interpolate.`**`make_lsq_spline`**(*x*, *y*, *t*, *k=3*, *w=None*, *axis=0*, *check_finite=True*)

Compute the (coefficients of) an LSQ B-spline.

The result is a linear combination

$$S(x) = \sum_j c_j B_j(x; t)$$

of the B-spline basis elements, $B_j(x; t)$, which minimizes

$$\sum_j \left(w_j \times (S(x_j) - y_j) \right)^2$$

Parameters

x : array_like, shape (m,)
Abscissas.

y : array_like, shape (m, ...)
Ordinates.

t : array_like, shape (n + k + 1,).
Knots. Knots and data points must satisfy Schoenberg-Whitney conditions.

k : int, optional
B-spline degree. Default is cubic, k=3.

w : array_like, shape (n,), optional
Weights for spline fitting. Must be positive. If `None`, then weights are all equal. Default is `None`.

axis : int, optional
Interpolation axis. Default is zero.

check_finite : bool, optional
Whether to check that the input arrays contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is True.

Returns

b : a BSpline object of the degree *k* with knots *t*.

See also:

`BSpline` base class representing the B-spline objects

`make_interp_spline` a similar factory function for interpolating splines

`LSQUnivariateSpline` a FITPACK-based spline fitting routine

`splrep` a FITPACK-based fitting routine

Notes

The number of data points must be larger than the spline degree *k*.

Knots *t* must satisfy the Schoenberg-Whitney conditions, i.e., there must be a subset of data points `x[j]` such that `t[j] < x[j] < t[j+k+1]`, for `j=0, 1,...,n-k-2`.

Examples

Generate some noisy data:

```
>>> x = np.linspace(-3, 3, 50)
>>> y = np.exp(-x**2) + 0.1 * np.random.randn(50)
```

Now fit a smoothing cubic spline with a pre-defined internal knots. Here we make the knot vector (k+1)-regular by adding boundary knots:

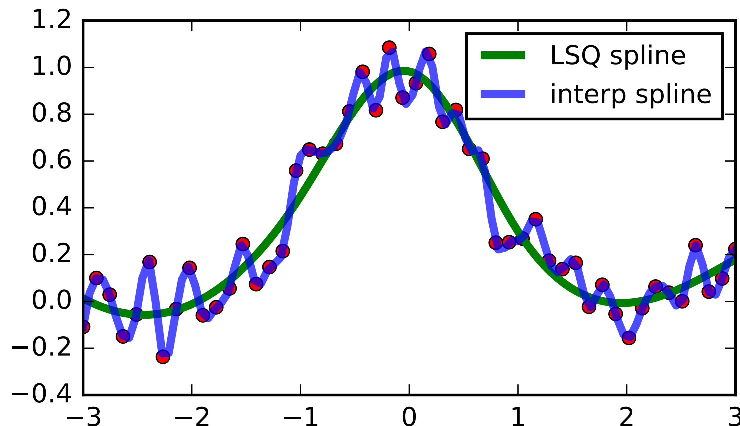
```
>>> from scipy.interpolate import make_lsq_spline, BSpline
>>> t = [-1, 0, 1]
>>> k = 3
>>> t = np.r_[(x[0],)*(k+1),
...           t,
...           (x[-1],)*(k+1)]
>>> spl = make_lsq_spline(x, y, t, k)
```

For comparison, we also construct an interpolating spline for the same set of data:

```
>>> from scipy.interpolate import make_interp_spline
>>> spl_i = make_interp_spline(x, y)
```

Plot both:

```
>>> import matplotlib.pyplot as plt
>>> xs = np.linspace(-3, 3, 100)
>>> plt.plot(x, y, 'ro', ms=5)
>>> plt.plot(xs, spl(xs), 'g-', lw=3, label='LSQ spline')
>>> plt.plot(xs, spl_i(xs), 'b-', lw=3, alpha=0.7, label='interp spline')
>>> plt.legend(loc='best')
>>> plt.show()
```



NaN handling: If the input arrays contain `nan` values, the result is not useful since the underlying spline fitting routines cannot deal with `nan`. A workaround is to use zero weights for not-a-number data points:

```
>>> y[8] = np.nan
>>> w = np.isnan(y)
>>> y[w] = 0.
>>> tck = make_lsq_spline(x, y, t, w=~w)
```

Notice the need to replace a `nan` by a numerical value (precise value does not matter as long as the corresponding weight is zero.)

Functional interface to FITPACK routines:

splrep(x, y[, w, xb, xe, k, task, s, t, ...])	Find the B-spline representation of 1-D curve.

Continued on next page

Table 5.47 – continued from previous page

splprep(x[, w, u, ub, ue, k, task, s, t, ...])	Find the B-spline representation of an N-dimensional curve.
splev(x, tck[, der, ext])	Evaluate a B-spline or its derivatives.
splint(a, b, tck[, full_output])	Evaluate the definite integral of a B-spline between two given points.
sproot(tck[, mest])	Find the roots of a cubic B-spline.
spalde(x, tck)	Evaluate all derivatives of a B-spline.
splder(tck[, n])	Compute the spline representation of the derivative of a given spline
splantider(tck[, n])	Compute the spline for the antiderivative (integral) of a given spline.
insert(x, tck[, m, per])	Insert knots into a B-spline.

scipy.interpolate.**splrep**(*x*, *y*, *w=None*, *xb=None*, *xe=None*, *k=3*, *task=0*, *s=None*, *t=None*, *full_output=0*, *per=0*, *quiet=1*)

Find the B-spline representation of 1-D curve.

Given the set of data points `(x[i], y[i])` determine a smooth spline approximation of degree k on the interval `xb <= x <= xe`.

Parameters

x, y : array_like
The data points defining a curve y = f(x).

w : array_like, optional
Strictly positive rank-1 array of weights the same length as x and y. The weights are used in computing the weighted least-squares spline fit. If the errors in the y values have standard-deviation given by the vector d, then w should be 1/d. Default is ones(len(x)).

xb, xe : float, optional
The interval to fit. If None, these default to x[0] and x[-1] respectively.

k : int, optional
The degree of the spline fit. It is recommended to use cubic splines. Even values of k should be avoided especially with small s values. 1 <= k <= 5

task : {1, 0, -1}, optional
If task==0 find t and c for a given smoothing factor, s.
If task==1 find t and c for another value of the smoothing factor, s. There must have been a previous call with task=0 or task=1 for the same set of data (t will be stored an used internally)
If task=-1 find the weighted least square spline for a given set of knots, t. These should be interior knots as knots on the ends will be added automatically.

s : float, optional
A smoothing condition. The amount of smoothness is determined by satisfying the conditions: sum((w * (y - g))**2,axis=0) <= s where g(x) is the smoothed interpolation of (x,y). The user can use s to control the tradeoff between closeness and smoothness of fit. Larger s means more smoothing while smaller values of s indicate less smoothing. Recommended values of s depend on the weights, w. If the weights represent the inverse of the standard-deviation of y, then a good s value should be found in the range (m-sqrt(2*m),m+sqrt(2*m)) where m is the number of datapoints in x, y, and w. default : s=m-sqrt(2*m) if weights are supplied. s = 0.0 (interpolating) if no weights are supplied.

t : array_like, optional
The knots needed for task=-1. If given then task is automatically set to -1.

full_output : bool, optional
If non-zero, then return optional outputs.

per : bool, optional

If non-zero, data points are considered periodic with period x[m-1] - x[0] and a smooth periodic spline approximation is returned. Values of y[m-1] and w[m-1] are not used.

quiet : bool, optional
Non-zero to suppress messages. This parameter is deprecated; use standard Python warning filters instead.

Returns

tck : tuple
A tuple (t,c,k) containing the vector of knots, the B-spline coefficients, and the degree of the spline.

fp : array, optional
The weighted sum of squared residuals of the spline approximation.

ier : int, optional
An integer flag about splrep success. Success is indicated if ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg : str, optional
A message corresponding to the integer flag, ier.

See also:

`UnivariateSpline`, *`BivariateSpline`*, *`splprep`*, *`splev`*, *`sproot`*, *`spalde`*, *`splint`*, *`bisplrep`*, *`bisplev`*, *`BSpline`*, *`make_interp_spline`*

Notes

See *`splev`* for evaluation of the spline and its derivatives. Uses the FORTRAN routine `curfit` from FITPACK.

The user is responsible for assuring that the values of *x* are unique. Otherwise, *`splrep`* will not return sensible results.

If provided, knots *t* must satisfy the Schoenberg-Whitney conditions, i.e., there must be a subset of data points `x[j]` such that `t[j] < x[j] < t[j+k+1]`, for `j=0, 1,...,n-k-2`.

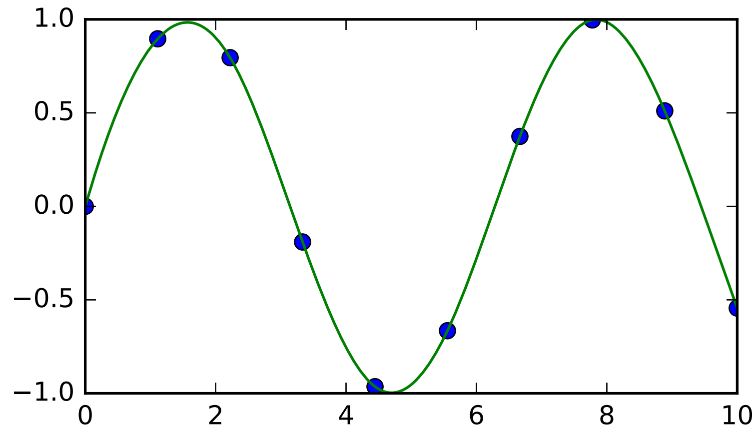
References

Based on algorithms described in *[R86]*, *[R87]*, *[R88]*, and *[R89]*:

[R86], *[R87]*, *[R88]*, *[R89]*

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy.interpolate import splev, splrep
>>> x = np.linspace(0, 10, 10)
>>> y = np.sin(x)
>>> spl = splrep(x, y)
>>> x2 = np.linspace(0, 10, 200)
>>> y2 = splev(x2, spl)
>>> plt.plot(x, y, 'o', x2, y2)
>>> plt.show()
```



`scipy.interpolate.splprep(x, w=None, u=None, ub=None, ue=None, k=3, task=0, s=None, t=None, full_output=0, nest=None, per=0, quiet=1)`

Find the B-spline representation of an N-dimensional curve.

Given a list of N rank-1 arrays, *x*, which represent a curve in N-dimensional space parametrized by *u*, find a smooth approximating spline curve g(*u*). Uses the FORTRAN routine parcur from FITPACK.

Parameters

x : array_like
: A list of sample vector arrays representing the curve.

w : array_like, optional
: Strictly positive rank-1 array of weights the same length as *x[0]*. The weights are used in computing the weighted least-squares spline fit. If the errors in the *x* values have standard-deviation given by the vector d, then *w* should be 1/d. Default is `ones(len(x[0]))`.

u : array_like, optional
: An array of parameter values. If not given, these values are calculated automatically as `M = len(x[0])`, where

  v[0] = 0

  v[i] = v[i-1] + distance(*x[i]*, *x[i-1]*)

  u[i] = v[i] / v[M-1]

ub, ue : int, optional
: The end-points of the parameters interval. Defaults to u[0] and u[-1].

k : int, optional
: Degree of the spline. Cubic splines are recommended. Even values of *k* should be avoided especially with a small s-value. `1 <= k <= 5`, default is 3.

task : int, optional
: If task==0 (default), find t and c for a given smoothing factor, s. If task==1, find t and c for another value of the smoothing factor, s. There must have been a previous call with task=0 or task=1 for the same set of data. If task=-1 find the weighted least square spline for a given set of knots, t.

s : float, optional
: A smoothing condition. The amount of smoothness is determined by satisfying the conditions: `sum((w * (y - g))**2,axis=0) <= s`, where g(x) is the smoothed interpolation of (x,y). The user can use `s` to control the trade-off between closeness and smoothness of fit. Larger `s` means more smoothing while smaller values of `s` indicate less smoothing. Recommended values of `s` depend on the weights, w. If the

weights represent the inverse of the standard-deviation of y, then a good `s` value should be found in the range `(m-sqrt(2*m),m+sqrt(2*m))`, where m is the number of data points in x, y, and w.

t : int, optional
The knots needed for task=-1.

full_output : int, optional
If non-zero, then return optional outputs.

nest : int, optional
An over-estimate of the total number of knots of the spline to help in determining the storage space. By default nest=m/2. Always large enough is nest=m+k+1.

per : int, optional
If non-zero, data points are considered periodic with period `x[m-1] - x[0]` and a smooth periodic spline approximation is returned. Values of `y[m-1]` and `w[m-1]` are not used.

quiet : int, optional
Non-zero to suppress messages. This parameter is deprecated; use standard Python warning filters instead.

Returns

tck : tuple
(t,c,k) a tuple containing the vector of knots, the B-spline coefficients, and the degree of the spline.

u : array
An array of the values of the parameter.

fp : float
The weighted sum of squared residuals of the spline approximation.

ier : int
An integer flag about splrep success. Success is indicated if ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg : str
A message corresponding to the integer flag, ier.

See also:

splrep, *splev*, *sproot*, *spalde*, *splint*, *bisplrep*, *bisplev*, UnivariateSpline, *BivariateSpline*, *BSpline*, *make_interp_spline*

Notes

See *splev* for evaluation of the spline and its derivatives. The number of dimensions N must be smaller than 11.

References

[R83], [R84], [R85]

Examples

Generate a discretization of a limacon curve in the polar coordinates:

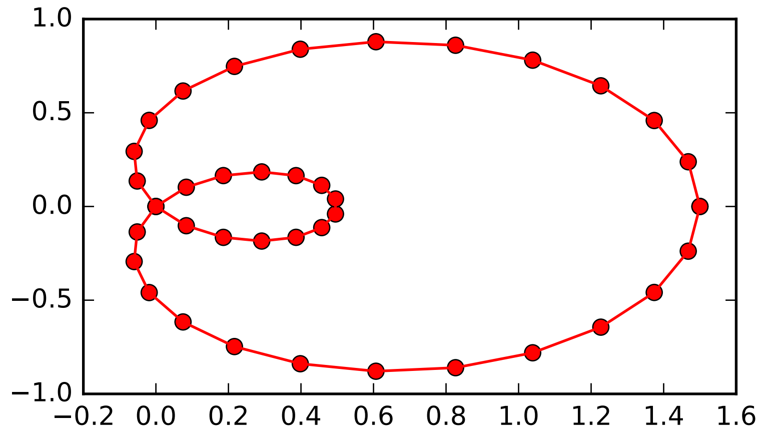
```
>>> phi = np.linspace(0, 2.*np.pi, 40)
>>> r = 0.5 + np.cos(phi)         # polar coords
>>> x, y = r * np.cos(phi), r * np.sin(phi)    # convert to cartesian
```

And interpolate:

```
>>> from scipy.interpolate import splprep, splev
>>> tck, u = splprep([x, y], s=0)
>>> new_points = splev(u, tck)
```

Notice that (i) we force interpolation by using *s=0*, (ii) the parameterization, `u`, is generated automatically. Now plot the result:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> ax.plot(x, y, 'ro')
>>> ax.plot(new_points[0], new_points[1], 'r-')
>>> plt.show()
```



`scipy.interpolate.`**`splev`**(*x*, *tck*, *der=0*, *ext=0*)

Evaluate a B-spline or its derivatives.

Given the knots and coefficients of a B-spline representation, evaluate the value of the smoothing polynomial and its derivatives. This is a wrapper around the FORTRAN routines splev and splder of FITPACK.

Parameters

x : array_like
: An array of points at which to return the value of the smoothed spline or its derivatives. If *tck* was returned from *`splprep`*, then the parameter values, u should be given.

tck : 3-tuple or a BSpline object
: If a tuple, then it should be a sequence of length 3 returned by *`splrep`* or *`splprep`* containing the knots, coefficients, and degree of the spline. (Also see Notes.)

der : int, optional
: The order of derivative of the spline to compute (must be less than or equal to k).

ext : int, optional
: Controls the value returned for elements of `x` not in the interval defined by the knot sequence.
  - if ext=0, return the extrapolated value.
  - if ext=1, return 0
  - if ext=2, raise a ValueError
  - if ext=3, return the boundary value.

  The default value is 0.

Returns

y : ndarray or list of ndarrays
: An array of values representing the spline function evaluated at the points in *x*. If *tck* was returned from *`splprep`*, then this is a list of arrays representing the curve in N-dimensional space.

See also:

splprep, *splrep*, *sproot*, *spalde*, *splint*, *bisplrep*, *bisplev*, *BSpline*

Notes

Manipulating the tck-tuples directly is not recommended. In new code, prefer using `BSpline` objects.

References

[R78], *[R79]*, *[R80]*

scipy.interpolate.**splint**(*a*, *b*, *tck*, *full_output=0*)

Evaluate the definite integral of a B-spline between two given points.

Parameters

a, b : float
The end-points of the integration interval.

tck : tuple or a BSpline instance
If a tuple, then it should be a sequence of length 3, containing the vector of knots, the B-spline coefficients, and the degree of the spline (see `splev`).

full_output : int, optional
Non-zero to return optional output.

Returns

integral : float
The resulting integral.

wrk : ndarray
An array containing the integrals of the normalized B-splines defined on the set of knots. (Only returned if *full_output* is non-zero)

See also:

splprep, *splrep*, *sproot*, *spalde*, *splev*, *bisplrep*, *bisplev*, *BSpline*

Notes

`splint` silently assumes that the spline function is zero outside the data interval (*a*, *b*).

Manipulating the tck-tuples directly is not recommended. In new code, prefer using the `BSpline` objects.

References

[R81], *[R82]*

scipy.interpolate.**sproot**(*tck*, *mest=10*)

Find the roots of a cubic B-spline.

Given the knots (>=8) and coefficients of a cubic B-spline return the roots of the spline.

Parameters

tck : tuple or a BSpline object
If a tuple, then it should be a sequence of length 3, containing the vector of knots, the B-spline coefficients, and the degree of the spline. The number of knots must be >= 8, and the degree must be 3. The knots must be a montonically increasing sequence.

mest : int, optional
An estimate of the number of zeros (Default is 10).

Returns

zeros : ndarray
An array giving the roots of the spline.

See also:

splprep, *splrep*, *splint*, *spalde*, *splev*, *bisplrep*, *bisplev*, *BSpline*

Notes

Manipulating the tck-tuples directly is not recommended. In new code, prefer using the `BSpline` objects.

References

[R90], [R91], [R92]

`scipy.interpolate.`**`spalde`**`(`*x*, *tck*`)`

Evaluate all derivatives of a B-spline.

Given the knots and coefficients of a cubic B-spline compute all derivatives up to order k at a point (or set of points).

Parameters

x : array_like
: A point or a set of points at which to evaluate the derivatives. Note that `t(k) <= x <= t(n-k+1)` must hold for each *x*.

tck : tuple
: A tuple `(t, c, k)`, containing the vector of knots, the B-spline coefficients, and the degree of the spline (see `splev`).

Returns

results : {ndarray, list of ndarrays}
: An array (or a list of arrays) containing all derivatives up to order k inclusive for each point *x*.

See also:

`splprep`, `splrep`, `splint`, `sproot`, `splev`, `bisplrep`, `bisplev`, `BSpline`

References

[R75], [R76], [R77]

`scipy.interpolate.`**`splder`**`(`*tck*, *n=1*`)`

Compute the spline representation of the derivative of a given spline

Parameters

tck : BSpline instance or a tuple of (t, c, k)
: Spline whose derivative to compute

n : int, optional
: Order of derivative to evaluate. Default: 1

Returns

`BSpline` instance or tuple
: Spline of order k2=k-n representing the derivative of the input spline. A tuple is returned iff the input argument *tck* is a tuple, otherwise a BSpline object is constructed and returned.

See also:

`splantider`, `splev`, `spalde`, `BSpline`

Notes

New in version 0.13.0.

Examples

This can be used for finding maxima of a curve:

```
>>> from scipy.interpolate import splrep, splder, sproot
>>> x = np.linspace(0, 10, 70)
>>> y = np.sin(x)
>>> spl = splrep(x, y, k=4)
```

Now, differentiate the spline and find the zeros of the derivative. (NB: `sproot` only works for order 3 splines, so we fit an order 4 spline):

```
>>> dspl = splder(spl)
>>> sproot(dspl) / np.pi
array([ 0.50000001,  1.5       ,  2.49999998])
```

This agrees well with roots $\pi/2 + n\pi$ of $\cos(x) = \sin'(x)$.

`scipy.interpolate.splantider(tck, n=1)`

Compute the spline for the antiderivative (integral) of a given spline.

Parameters
- **tck** : BSpline instance or a tuple of (t, c, k)
  Spline whose antiderivative to compute
- **n** : int, optional
  Order of antiderivative to evaluate. Default: 1

Returns
BSpline instance or a tuple of (t2, c2, k2)
Spline of order k2=k+n representing the antiderivative of the input spline. A tuple is returned iff the input argument *tck* is a tuple, otherwise a BSpline object is constructed and returned.

See also:

splder, *splev*, *spalde*, *BSpline*

Notes

The *splder* function is the inverse operation of this function. Namely, `splder(splantider(tck))` is identical to *tck*, modulo rounding error.

New in version 0.13.0.

Examples

```
>>> from scipy.interpolate import splrep, splder, splantider, splev
>>> x = np.linspace(0, np.pi/2, 70)
>>> y = 1 / np.sqrt(1 - 0.8*np.sin(x)**2)
>>> spl = splrep(x, y)
```

The derivative is the inverse operation of the antiderivative, although some floating point error accumulates:

```
>>> splev(1.7, spl), splev(1.7, splder(splantider(spl)))
(array(2.1565429877197317), array(2.1565429877201865))
```

Antiderivative can be used to evaluate definite integrals:

```
>>> ispl = splantider(spl)
>>> splev(np.pi/2, ispl) - splev(0, ispl)
2.2572053588768486
```

This is indeed an approximation to the complete elliptic integral $K(m) = \int_0^{\pi/2} [1 - m \sin^2 x]^{-1/2} dx$:

```
>>> from scipy.special import ellipk
>>> ellipk(0.8)
2.2572053268208538
```

`scipy.interpolate.insert(x, tck, m=1, per=0)`

Insert knots into a B-spline.

Given the knots and coefficients of a B-spline representation, create a new B-spline with a knot inserted *m* times at point *x*. This is a wrapper around the FORTRAN routine insert of FITPACK.

Parameters
- **x (u)** : array_like

A 1-D point at which to insert a new knot(s). If *tck* was returned from `splprep`, then the parameter values, u should be given.

tck : a *BSpline* instance or a tuple
: If tuple, then it is expected to be a tuple (t,c,k) containing the vector of knots, the B-spline coefficients, and the degree of the spline.

m : int, optional
: The number of times to insert the given knot (its multiplicity). Default is 1.

per : int, optional
: If non-zero, the input spline is considered periodic.

Returns BSpline instance or a tuple
: A new B-spline with knots t, coefficients c, and degree k. `t(k+1) <= x <= t(n-k)`, where k is the degree of the spline. In case of a periodic spline (`per != 0`) there must be either at least k interior knots t(j) satisfying `t(k+1)<t(j)<=x` or at least k interior knots t(j) satisfying `x<=t(j)<t(n-k)`. A tuple is returned iff the input argument *tck* is a tuple, otherwise a BSpline object is constructed and returned.

Notes

Based on algorithms from *[R73]* and *[R74]*.

Manipulating the tck-tuples directly is not recommended. In new code, prefer using the *BSpline* objects.

References

[R73], *[R74]*

Object-oriented FITPACK interface:

`UnivariateSpline`(x, y[, w, bbox, k, s, ext, ...])	One-dimensional smoothing spline fit to a given set of data points.
`InterpolatedUnivariateSpline`(x, y[, w, ...])	One-dimensional interpolating spline for a given set of data points.
`LSQUnivariateSpline`(x, y, t[, w, bbox, k, ...])	One-dimensional spline with explicit internal knots.

5.7.4 2-D Splines

For data on a grid:

RectBivariateSpline(x, y, z[, bbox, kx, ky, s])	Bivariate spline approximation over a rectangular mesh.
RectSphereBivariateSpline(u, v, r[, s, ...])	Bivariate spline approximation over a rectangular mesh on a sphere.

class `scipy.interpolate.`**`RectSphereBivariateSpline`**(*u*, *v*, *r*, *s=0.0*, *pole_continuity=False*, *pole_values=None*, *pole_exact=False*, *pole_flat=False*)

Bivariate spline approximation over a rectangular mesh on a sphere.

Can be used for smoothing data.

New in version 0.11.0.

Parameters **u** : array_like
: 1-D array of latitude coordinates in strictly ascending order. Coordinates must be given in radians and lie within the interval (0, pi).

v : array_like

1-D array of longitude coordinates in strictly ascending order. Coordinates must be given in radians. First element (v[0]) must lie within the interval [-pi, pi). Last element (v[-1]) must satisfy v[-1] <= v[0] + 2*pi.

r : array_like
: 2-D array of data with shape `(u.size, v.size)`.

s : float, optional
: Positive smoothing factor defined for estimation condition (`s=0` is for interpolation).

pole_continuity : bool or (bool, bool), optional
: Order of continuity at the poles `u=0` (`pole_continuity[0]`) and `u=pi` (`pole_continuity[1]`). The order of continuity at the pole will be 1 or 0 when this is True or False, respectively. Defaults to False.

pole_values : float or (float, float), optional
: Data values at the poles `u=0` and `u=pi`. Either the whole parameter or each individual element can be None. Defaults to None.

pole_exact : bool or (bool, bool), optional
: Data value exactness at the poles `u=0` and `u=pi`. If True, the value is considered to be the right function value, and it will be fitted exactly. If False, the value will be considered to be a data value just like the other data values. Defaults to False.

pole_flat : bool or (bool, bool), optional
: For the poles at `u=0` and `u=pi`, specify whether or not the approximation has vanishing derivatives. Defaults to False.

See also:

`RectBivariateSpline`
: bivariate spline approximation over a rectangular mesh

Notes

Currently, only the smoothing spline approximation (`iopt[0] = 0` and `iopt[0] = 1` in the FITPACK routine) is supported. The exact least-squares spline approximation is not implemented yet.

When actually performing the interpolation, the requested *v* values must lie within the same length 2pi interval that the original *v* values were chosen from.

For more information, see the FITPACK site about this function.

Examples

Suppose we have global data on a coarse grid

```
>>> lats = np.linspace(10, 170, 9) * np.pi / 180.
>>> lons = np.linspace(0, 350, 18) * np.pi / 180.
>>> data = np.dot(np.atleast_2d(90. - np.linspace(-80., 80., 18)).T,
...               np.atleast_2d(180. - np.abs(np.linspace(0., 350., 9)))).T
```

We want to interpolate it to a global one-degree grid

```
>>> new_lats = np.linspace(1, 180, 180) * np.pi / 180
>>> new_lons = np.linspace(1, 360, 360) * np.pi / 180
>>> new_lats, new_lons = np.meshgrid(new_lats, new_lons)
```

We need to set up the interpolator object

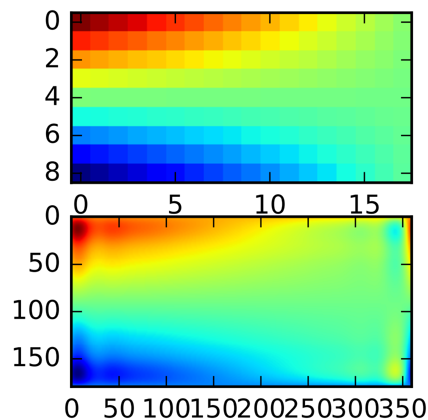
```
>>> from scipy.interpolate import RectSphereBivariateSpline
>>> lut = RectSphereBivariateSpline(lats, lons, data)
```

Finally we interpolate the data. The *RectSphereBivariateSpline* object only takes 1-D arrays as input, therefore we need to do some reshaping.

```
>>> data_interp = lut.ev(new_lats.ravel(),
...                      new_lons.ravel()).reshape((360, 180)).T
```

Looking at the original and the interpolated data, one can see that the interpolant reproduces the original data very well:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(211)
>>> ax1.imshow(data, interpolation='nearest')
>>> ax2 = fig.add_subplot(212)
>>> ax2.imshow(data_interp, interpolation='nearest')
>>> plt.show()
```



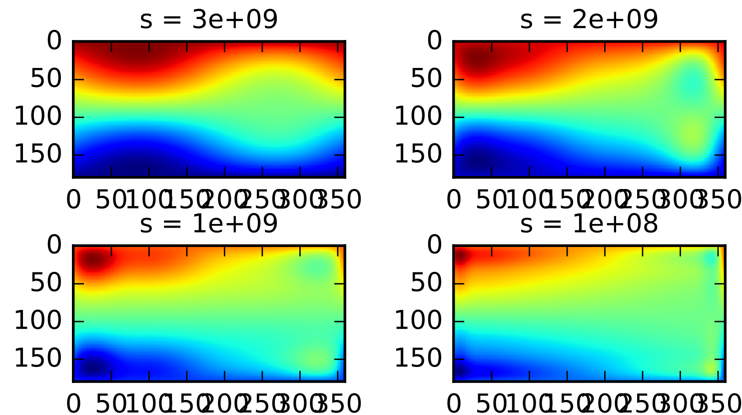
Chosing the optimal value of `s` can be a delicate task. Recommended values for `s` depend on the accuracy of the data values. If the user has an idea of the statistical errors on the data, she can also find a proper estimate for `s`. By assuming that, if she specifies the right `s`, the interpolator will use a spline `f(u,v)` which exactly reproduces the function underlying the data, she can evaluate `sum((r(i,j)-s(u(i),v(j)))**2)` to find a good estimate for this `s`. For example, if she knows that the statistical errors on her `r(i,j)`-values are not greater than 0.1, she may expect that a good `s` should have a value not larger than `u.size * v.size * (0.1)**2`.

If nothing is known about the statistical error in `r(i,j)`, `s` must be determined by trial and error. The best is then to start with a very large value of `s` (to determine the least-squares polynomial and the corresponding upper bound `fp0` for `s`) and then to progressively decrease the value of `s` (say by a factor 10 in the beginning, i.e. `s = fp0 / 10, fp0 / 100, ...` and more carefully as the approximation shows more detail) to obtain closer fits.

The interpolation results for different values of `s` give some insight into this process:

```
>>> fig2 = plt.figure()
>>> s = [3e9, 2e9, 1e9, 1e8]
>>> for ii in xrange(len(s)):
...     lut = RectSphereBivariateSpline(lats, lons, data, s=s[ii])
...     data_interp = lut.ev(new_lats.ravel(),
```

```
...                          new_lons.ravel()).reshape((360, 180)).T
...     ax = fig2.add_subplot(2, 2, ii+1)
...     ax.imshow(data_interp, interpolation='nearest')
...     ax.set_title("s = %g" % s[ii])
>>> plt.show()
```



Methods

`__call__`(theta, phi[, dtheta, dphi, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(theta, phi[, dtheta, dphi])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.
`get_knots`()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
`get_residual`()	Return weighted sum of squared residuals of the spline

`RectSphereBivariateSpline.`**`__call__`**(*theta*, *phi*, *dtheta=0*, *dphi=0*, *grid=True*)

Evaluate the spline or its derivatives at given positions.

Parameters

theta, phi : array_like

Input coordinates.

If *grid* is False, evaluate the spline at points `(theta[i], phi[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.

If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays theta, phi. The arrays must be sorted to increasing order.

dtheta : int, optional

Order of theta-derivative

New in version 0.14.0.

dphi : int

Order of phi-derivative

New in version 0.14.0.

grid : bool

Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.

New in version 0.14.0.

`RectSphereBivariateSpline.`**`ev`**`(`*theta*, *phi*, *dtheta=0*, *dphi=0*`)`

Evaluate the spline at points

Returns the interpolated value at `(theta[i], phi[i]), i=0,...,len(theta)-1`.

Parameters **theta, phi** : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.
dtheta : int, optional
Order of theta-derivative
New in version 0.14.0.
dphi : int, optional
Order of phi-derivative
New in version 0.14.0.

`RectSphereBivariateSpline.`**`get_coeffs`**`()`

Return spline coefficients.

`RectSphereBivariateSpline.`**`get_knots`**`()`

Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`RectSphereBivariateSpline.`**`get_residual`**`()`

Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

For unstructured data:

BivariateSpline	Base class for bivariate splines.
SmoothBivariateSpline(x, y, z[, w, bbox, ...])	Smooth bivariate spline approximation.
SmoothSphereBivariateSpline(theta, phi, r[, ...])	Smooth bivariate spline approximation in spherical coordinates.
LSQBivariateSpline(x, y, z, tx, ty[, w, ...])	Weighted least-squares bivariate spline approximation.
LSQSphereBivariateSpline(theta, phi, r, tt, tp)	Weighted least-squares bivariate spline approximation in spherical coordinates.

class `scipy.interpolate.`**`BivariateSpline`**

Base class for bivariate splines.

This describes a spline `s(x, y)` of degrees `kx` and `ky` on the rectangle `[xb, xe] * [yb, ye]` calculated from a given set of data points `(x, y, z)`.

This class is meant to be subclassed, not instantiated directly. To construct these splines, call either *`SmoothBivariateSpline`* or *`LSQBivariateSpline`*.

See also:

`UnivariateSpline`
a similar class for univariate spline interpolation

`SmoothBivariateSpline`
to create a BivariateSpline through the given points

`LSQBivariateSpline`
to create a BivariateSpline using weighted least-squares fitting

`SphereBivariateSpline`
bivariate spline interpolation in spherical cooridinates

`bisplrep` older wrapping of FITPACK

bisplev older wrapping of FITPACK

Methods

`__call__`(x, y[, mth, dx, dy, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(xi, yi[, dx, dy])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.
`get_knots`()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
`get_residual`()	Return weighted sum of squared residuals of the spline
`integral`(xa, xb, ya, yb)	Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

`BivariateSpline.`**`__call__`**(*x*, *y*, *mth=None*, *dx=0*, *dy=0*, *grid=True*)

Evaluate the spline or its derivatives at given positions.

Parameters

x, y : array_like
Input coordinates.
If *grid* is False, evaluate the spline at points `(x[i], y[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.
If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays x, y. The arrays must be sorted to increasing order.

dx : int
Order of x-derivative
New in version 0.14.0.

dy : int
Order of y-derivative
New in version 0.14.0.

grid : bool
Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.
New in version 0.14.0.

mth : str
Deprecated argument. Has no effect.

`BivariateSpline.`**`ev`**(*xi*, *yi*, *dx=0*, *dy=0*)

Evaluate the spline at points

Returns the interpolated value at `(xi[i], yi[i]), i=0,...,len(xi)-1`.

Parameters

xi, yi : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.

dx : int, optional
Order of x-derivative
New in version 0.14.0.

dy : int, optional
Order of y-derivative
New in version 0.14.0.

`BivariateSpline.`**`get_coeffs`**()

Return spline coefficients.

`BivariateSpline.`**`get_knots`**()

Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`BivariateSpline.`**`get_residual`**`()`

Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

`BivariateSpline.`**`integral`**`(`*xa*, *xb*, *ya*, *yb*`)`

Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

Parameters

xa, xb : float
: The end-points of the x integration interval.

ya, yb : float
: The end-points of the y integration interval.

Returns

integ : float
: The value of the resulting integral.

class `scipy.interpolate.`**`SmoothBivariateSpline`**`(`*x, y, z, w=None, bbox=[None, None, None, None], kx=3, ky=3, s=None, eps=None*`)`

Smooth bivariate spline approximation.

Parameters

x, y, z : array_like
: 1-D sequences of data points (order is not important).

w : array_like, optional
: Positive 1-D sequence of weights, of same length as *x*, *y* and *z*.

bbox : array_like, optional
: Sequence of length 4 specifying the boundary of the rectangular approximation domain. By default, `bbox=[min(x,tx),max(x,tx), min(y,ty),max(y,ty)]`.

kx, ky : ints, optional
: Degrees of the bivariate spline. Default is 3.

s : float, optional
: Positive smoothing factor defined for estimation condition: `sum((w[i]*(z[i]-s(x[i], y[i])))**2, axis=0) <= s` Default `s=len(w)` which should be a good value if `1/w[i]` is an estimate of the standard deviation of `z[i]`.

eps : float, optional
: A threshold for determining the effective rank of an over-determined linear system of equations. *eps* should have a value between 0 and 1, the default is 1e-16.

See also:

`bisplrep` an older wrapping of FITPACK

`bisplev` an older wrapping of FITPACK

`UnivariateSpline`
a similar class for univariate spline interpolation

`LSQUnivariateSpline`
to create a BivariateSpline using weighted

Notes

The length of *x*, *y* and *z* should be at least `(kx+1) * (ky+1)`.

Methods

`__call__`(x, y[, mth, dx, dy, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(xi, yi[, dx, dy])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.

Continued on next page

Table 5.53 – continued from previous page

get_knots()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
get_residual()	Return weighted sum of squared residuals of the spline
integral(xa, xb, ya, yb)	Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

SmoothBivariateSpline.**__call__**(*x*, *y*, *mth=None*, *dx=0*, *dy=0*, *grid=True*)

Evaluate the spline or its derivatives at given positions.

Parameters **x, y** : array_like

Input coordinates.

If *grid* is False, evaluate the spline at points `(x[i], y[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.

If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays x, y. The arrays must be sorted to increasing order.

dx : int

Order of x-derivative

New in version 0.14.0.

dy : int

Order of y-derivative

New in version 0.14.0.

grid : bool

Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.

New in version 0.14.0.

mth : str

Deprecated argument. Has no effect.

SmoothBivariateSpline.**ev**(*xi*, *yi*, *dx=0*, *dy=0*)

Evaluate the spline at points

Returns the interpolated value at `(xi[i], yi[i]), i=0,...,len(xi)-1`.

Parameters **xi, yi** : array_like

Input coordinates. Standard Numpy broadcasting is obeyed.

dx : int, optional

Order of x-derivative

New in version 0.14.0.

dy : int, optional

Order of y-derivative

New in version 0.14.0.

SmoothBivariateSpline.**get_coeffs**()

Return spline coefficients.

SmoothBivariateSpline.**get_knots**()

Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

SmoothBivariateSpline.**get_residual**()

Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

SmoothBivariateSpline.**integral**(*xa*, *xb*, *ya*, *yb*)

Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

Parameters **xa, xb** : float
The end-points of the x integration interval.
ya, yb : float
The end-points of the y integration interval.

Returns **integ** : float
The value of the resulting integral.

class `scipy.interpolate.SmoothSphereBivariateSpline`(*theta*, *phi*, *r*, *w=None*, *s=0.0*, *eps=1e-16*)

Smooth bivariate spline approximation in spherical coordinates.

New in version 0.11.0.

Parameters **theta, phi, r** : array_like
1-D sequences of data points (order is not important). Coordinates must be given in radians. Theta must lie within the interval (0, pi), and phi must lie within the interval (0, 2pi).
w : array_like, optional
Positive 1-D sequence of weights.
s : float, optional
Positive smoothing factor defined for estimation condition: `sum((w(i)*(r(i) - s(theta(i), phi(i))))**2, axis=0) <= s` Default `s=len(w)` which should be a good value if 1/w[i] is an estimate of the standard deviation of r[i].
eps : float, optional
A threshold for determining the effective rank of an over-determined linear system of equations. *eps* should have a value between 0 and 1, the default is 1e-16.

Notes

For more information, see the FITPACK site about this function.

Examples

Suppose we have global data on a coarse grid (the input data does not have to be on a grid):

```
>>> theta = np.linspace(0., np.pi, 7)
>>> phi = np.linspace(0., 2*np.pi, 9)
>>> data = np.empty((theta.shape[0], phi.shape[0]))
>>> data[:,0], data[0,:], data[-1,:] = 0., 0., 0.
>>> data[1:-1,1], data[1:-1,-1] = 1., 1.
>>> data[1,1:-1], data[-2,1:-1] = 1., 1.
>>> data[2:-2,2], data[2:-2,-2] = 2., 2.
>>> data[2,2:-2], data[-3,2:-2] = 2., 2.
>>> data[3,3:-2] = 3.
>>> data = np.roll(data, 4, 1)
```

We need to set up the interpolator object

```
>>> lats, lons = np.meshgrid(theta, phi)
>>> from scipy.interpolate import SmoothSphereBivariateSpline
>>> lut = SmoothSphereBivariateSpline(lats.ravel(), lons.ravel(),
...                                   data.T.ravel(), s=3.5)
```

As a first test, we'll see what the algorithm returns when run on the input coordinates

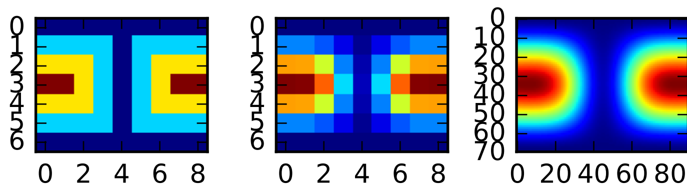
```
>>> data_orig = lut(theta, phi)
```

Finally we interpolate the data to a finer grid

```
>>> fine_lats = np.linspace(0., np.pi, 70)
>>> fine_lons = np.linspace(0., 2 * np.pi, 90)
```

```
>>> data_smth = lut(fine_lats, fine_lons)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(131)
>>> ax1.imshow(data, interpolation='nearest')
>>> ax2 = fig.add_subplot(132)
>>> ax2.imshow(data_orig, interpolation='nearest')
>>> ax3 = fig.add_subplot(133)
>>> ax3.imshow(data_smth, interpolation='nearest')
>>> plt.show()
```



Methods

`__call__`(theta, phi[, dtheta, dphi, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(theta, phi[, dtheta, dphi])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.
`get_knots`()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
`get_residual`()	Return weighted sum of squared residuals of the spline

`SmoothSphereBivariateSpline.`**`__call__`**(*theta*, *phi*, *dtheta=0*, *dphi=0*, *grid=True*)

Evaluate the spline or its derivatives at given positions.

Parameters **theta, phi** : array_like

Input coordinates.

If *grid* is False, evaluate the spline at points `(theta[i], phi[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.

If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays theta, phi. The arrays must be sorted to increasing order.

dtheta : int, optional

Order of theta-derivative
New in version 0.14.0.

dphi : int
Order of phi-derivative
New in version 0.14.0.

grid : bool
Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.
New in version 0.14.0.

`SmoothSphereBivariateSpline.`**`ev`**`(`*theta*, *phi*, *dtheta=0*, *dphi=0*`)`
Evaluate the spline at points

Returns the interpolated value at `(theta[i], phi[i]), i=0,...,len(theta)-1`.

Parameters

theta, phi : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.

dtheta : int, optional
Order of theta-derivative
New in version 0.14.0.

dphi : int, optional
Order of phi-derivative
New in version 0.14.0.

`SmoothSphereBivariateSpline.`**`get_coeffs`**`()`
Return spline coefficients.

`SmoothSphereBivariateSpline.`**`get_knots`**`()`
Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`SmoothSphereBivariateSpline.`**`get_residual`**`()`
Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

class `scipy.interpolate.`**`LSQBivariateSpline`**`(`*x, y, z, tx, ty, w=None, bbox=[None, None, None, None], kx=3, ky=3, eps=None*`)`
Weighted least-squares bivariate spline approximation.

Parameters

x, y, z : array_like
1-D sequences of data points (order is not important).

tx, ty : array_like
Strictly ordered 1-D sequences of knots coordinates.

w : array_like, optional
Positive 1-D array of weights, of the same length as *x*, *y* and *z*.

bbox : (4,) array_like, optional
Sequence of length 4 specifying the boundary of the rectangular approximation domain. By default, `bbox=[min(x,tx),max(x,tx), min(y,ty),max(y,ty)]`.

kx, ky : ints, optional
Degrees of the bivariate spline. Default is 3.

eps : float, optional
A threshold for determining the effective rank of an over-determined linear system of equations. *eps* should have a value between 0 and 1, the default is 1e-16.

See also:

`bisplrep` an older wrapping of FITPACK

bisplev an older wrapping of FITPACK

UnivariateSpline
a similar class for univariate spline interpolation

SmoothBivariateSpline
create a smoothing BivariateSpline

Notes

The length of *x*, *y* and *z* should be at least `(kx+1) * (ky+1)`.

Methods

__call__(x, y[, mth, dx, dy, grid])	Evaluate the spline or its derivatives at given positions.
ev(xi, yi[, dx, dy])	Evaluate the spline at points
get_coeffs()	Return spline coefficients.
get_knots()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
get_residual()	Return weighted sum of squared residuals of the spline
integral(xa, xb, ya, yb)	Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

LSQBivariateSpline.**__call__**(*x*, *y*, *mth=None*, *dx=0*, *dy=0*, *grid=True*)
Evaluate the spline or its derivatives at given positions.

Parameters

x, y : array_like
Input coordinates.
If *grid* is False, evaluate the spline at points `(x[i], y[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.
If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays x, y. The arrays must be sorted to increasing order.

dx : int
Order of x-derivative
New in version 0.14.0.

dy : int
Order of y-derivative
New in version 0.14.0.

grid : bool
Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.
New in version 0.14.0.

mth : str
Deprecated argument. Has no effect.

LSQBivariateSpline.**ev**(*xi*, *yi*, *dx=0*, *dy=0*)
Evaluate the spline at points

Returns the interpolated value at `(xi[i], yi[i]), i=0,...,len(xi)-1`.

Parameters

xi, yi : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.

dx : int, optional
Order of x-derivative
New in version 0.14.0.

dy : int, optional

Order of y-derivative
New in version 0.14.0.

`LSQBivariateSpline.get_coeffs()`
Return spline coefficients.

`LSQBivariateSpline.get_knots()`
Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`LSQBivariateSpline.get_residual()`
Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

`LSQBivariateSpline.integral(xa, xb, ya, yb)`
Evaluate the integral of the spline over area [xa,xb] x [ya,yb].

Parameters
xa, xb : float
The end-points of the x integration interval.
ya, yb : float
The end-points of the y integration interval.

Returns
integ : float
The value of the resulting integral.

class `scipy.interpolate.LSQSphereBivariateSpline(theta, phi, r, tt, tp, w=None, eps=1e-16)`
Weighted least-squares bivariate spline approximation in spherical coordinates.

New in version 0.11.0.

Parameters
theta, phi, r : array_like
1-D sequences of data points (order is not important). Coordinates must be given in radians. Theta must lie within the interval (0, pi), and phi must lie within the interval (0, 2pi).
tt, tp : array_like
Strictly ordered 1-D sequences of knots coordinates. Coordinates must satisfy `0 < tt[i] < pi`, `0 < tp[i] < 2*pi`.
w : array_like, optional
Positive 1-D sequence of weights, of the same length as *theta*, *phi* and *r*.
eps : float, optional
A threshold for determining the effective rank of an over-determined linear system of equations. *eps* should have a value between 0 and 1, the default is 1e-16.

Notes

For more information, see the FITPACK site about this function.

Examples

Suppose we have global data on a coarse grid (the input data does not have to be on a grid):

```
>>> theta = np.linspace(0., np.pi, 7)
>>> phi = np.linspace(0., 2*np.pi, 9)
>>> data = np.empty((theta.shape[0], phi.shape[0]))
>>> data[:,0], data[0,:], data[-1,:] = 0., 0., 0.
>>> data[1:-1,1], data[1:-1,-1] = 1., 1.
>>> data[1,1:-1], data[-2,1:-1] = 1., 1.
>>> data[2:-2,2], data[2:-2,-2] = 2., 2.
>>> data[2,2:-2], data[-3,2:-2] = 2., 2.
```

```
>>> data[3,3:-2] = 3.
>>> data = np.roll(data, 4, 1)
```

We need to set up the interpolator object. Here, we must also specify the coordinates of the knots to use.

```
>>> lats, lons = np.meshgrid(theta, phi)
>>> knotst, knotsp = theta.copy(), phi.copy()
>>> knotst[0] += .0001
>>> knotst[-1] -= .0001
>>> knotsp[0] += .0001
>>> knotsp[-1] -= .0001
>>> from scipy.interpolate import LSQSphereBivariateSpline
>>> lut = LSQSphereBivariateSpline(lats.ravel(), lons.ravel(),
...                                  data.T.ravel(), knotst, knotsp)
```

As a first test, we'll see what the algorithm returns when run on the input coordinates

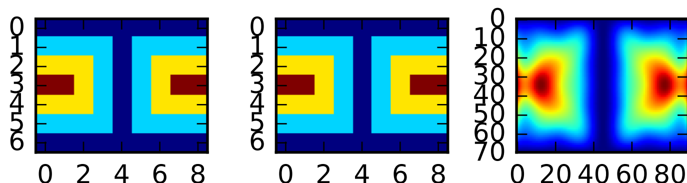
```
>>> data_orig = lut(theta, phi)
```

Finally we interpolate the data to a finer grid

```
>>> fine_lats = np.linspace(0., np.pi, 70)
>>> fine_lons = np.linspace(0., 2*np.pi, 90)
```

```
>>> data_lsq = lut(fine_lats, fine_lons)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(131)
>>> ax1.imshow(data, interpolation='nearest')
>>> ax2 = fig.add_subplot(132)
>>> ax2.imshow(data_orig, interpolation='nearest')
>>> ax3 = fig.add_subplot(133)
>>> ax3.imshow(data_lsq, interpolation='nearest')
>>> plt.show()
```



Methods

`__call__`(theta, phi[, dtheta, dphi, grid])	Evaluate the spline or its derivatives at given positions.
`ev`(theta, phi[, dtheta, dphi])	Evaluate the spline at points
`get_coeffs`()	Return spline coefficients.
`get_knots`()	Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively.
`get_residual`()	Return weighted sum of squared residuals of the spline

`LSQSphereBivariateSpline.`**`__call__`**(*theta*, *phi*, *dtheta=0*, *dphi=0*, *grid=True*)

Evaluate the spline or its derivatives at given positions.

Parameters

theta, phi : array_like
Input coordinates.
If *grid* is False, evaluate the spline at points `(theta[i], phi[i]), i=0, ..., len(x)-1`. Standard Numpy broadcasting is obeyed.
If *grid* is True: evaluate spline at the grid points defined by the coordinate arrays theta, phi. The arrays must be sorted to increasing order.

dtheta : int, optional
Order of theta-derivative
New in version 0.14.0.

dphi : int
Order of phi-derivative
New in version 0.14.0.

grid : bool
Whether to evaluate the results on a grid spanned by the input arrays, or at points specified by the input arrays.
New in version 0.14.0.

`LSQSphereBivariateSpline.`**`ev`**(*theta*, *phi*, *dtheta=0*, *dphi=0*)

Evaluate the spline at points

Returns the interpolated value at `(theta[i], phi[i]), i=0,...,len(theta)-1`.

Parameters

theta, phi : array_like
Input coordinates. Standard Numpy broadcasting is obeyed.

dtheta : int, optional
Order of theta-derivative
New in version 0.14.0.

dphi : int, optional
Order of phi-derivative
New in version 0.14.0.

`LSQSphereBivariateSpline.`**`get_coeffs`**()

Return spline coefficients.

`LSQSphereBivariateSpline.`**`get_knots`**()

Return a tuple (tx,ty) where tx,ty contain knots positions of the spline with respect to x-, y-variable, respectively. The position of interior and additional knots are given as t[k+1:-k-1] and t[:k+1]=b, t[-k-1:]=e, respectively.

`LSQSphereBivariateSpline.`**`get_residual`**()

Return weighted sum of squared residuals of the spline approximation: sum ((w[i]*(z[i]-s(x[i],y[i])))**2,axis=0)

Low-level interface to FITPACK functions:

bisplrep(x, y, z[, w, xb, xe, yb, ye, kx, ...])	Find a bivariate B-spline representation of a surface.
bisplev(x, y, tck[, dx, dy])	Evaluate a bivariate B-spline and its derivatives.

scipy.interpolate.**bisplrep**(*x, y, z, w=None, xb=None, xe=None, yb=None, ye=None, kx=3, ky=3, task=0, s=None, eps=1e-16, tx=None, ty=None, full_output=0, nxest=None, nyest=None, quiet=1*)

Find a bivariate B-spline representation of a surface.

Given a set of data points (x[i], y[i], z[i]) representing a surface z=f(x,y), compute a B-spline representation of the surface. Based on the routine SURFIT from FITPACK.

Parameters

x, y, z : ndarray
Rank-1 arrays of data points.

w : ndarray, optional
Rank-1 array of weights. By default `w=np.ones(len(x))`.

xb, xe : float, optional
End points of approximation interval in *x*. By default `xb = x.min(), xe=x.max()`.

yb, ye : float, optional
End points of approximation interval in *y*. By default `yb=y.min(), ye = y.max()`.

kx, ky : int, optional
The degrees of the spline (1 <= kx, ky <= 5). Third order (kx=ky=3) is recommended.

task : int, optional
If task=0, find knots in x and y and coefficients for a given smoothing factor, s. If task=1, find knots and coefficients for another value of the smoothing factor, s. bisplrep must have been previously called with task=0 or task=1. If task=-1, find coefficients for a given set of knots tx, ty.

s : float, optional
A non-negative smoothing factor. If weights correspond to the inverse of the standard-deviation of the errors in z, then a good s-value should be found in the range `(m-sqrt(2*m),m+sqrt(2*m))` where m=len(x).

eps : float, optional
A threshold for determining the effective rank of an over-determined linear system of equations (0 < eps < 1). *eps* is not likely to need changing.

tx, ty : ndarray, optional
Rank-1 arrays of the knots of the spline for task=-1

full_output : int, optional
Non-zero to return optional outputs.

nxest, nyest : int, optional
Over-estimates of the total number of knots. If None then `nxest = max(kx+sqrt(m/2),2*kx+3), nyest = max(ky+sqrt(m/2),2*ky+3)`.

quiet : int, optional
Non-zero to suppress printing of messages. This parameter is deprecated; use standard Python warning filters instead.

Returns

tck : array_like
A list [tx, ty, c, kx, ky] containing the knots (tx, ty) and coefficients (c) of the bivariate B-spline representation of the surface along with the degree of the spline.

fp : ndarray
The weighted sum of squared residuals of the spline approximation.

ier : int
An integer flag about splrep success. Success is indicated if ier<=0. If ier in [1,2,3] an error occurred but was not raised. Otherwise an error is raised.

msg : str
A message corresponding to the integer flag, ier.

See also:

splprep, *splrep*, *splint*, *sproot*, *splev*, UnivariateSpline, *BivariateSpline*

Notes

See *bisplev* to evaluate the value of the B-spline given its tck representation.

References

[R70], *[R71]*, *[R72]*

`scipy.interpolate.`**`bisplev`**(*x*, *y*, *tck*, *dx=0*, *dy=0*)
Evaluate a bivariate B-spline and its derivatives.

Return a rank-2 array of spline function values (or spline derivative values) at points given by the cross-product of the rank-1 arrays *x* and *y*. In special cases, return an array or just a float if either *x* or *y* or both are floats. Based on BISPEV from FITPACK.

Parameters

x, y : ndarray
Rank-1 arrays specifying the domain over which to evaluate the spline or its derivative.

tck : tuple
A sequence of length 5 returned by *bisplrep* containing the knot locations, the coefficients, and the degree of the spline: [tx, ty, c, kx, ky].

dx, dy : int, optional
The orders of the partial derivatives in *x* and *y* respectively.

Returns

vals : ndarray
The B-spline or its derivative evaluated over the set formed by the cross-product of *x* and *y*.

See also:

splprep, *splrep*, *splint*, *sproot*, *splev*, UnivariateSpline, *BivariateSpline*

Notes

See *bisplrep* to generate the *tck* representation.

References

[R67], *[R68]*, *[R69]*

5.7.5 Additional tools

lagrange(x, w)	Return a Lagrange interpolating polynomial.
approximate_taylor_polynomial(f, x, degree, ...)	Estimate the Taylor polynomial of f at x by polynomial fitting.
pade(an, m)	Return Pade approximation to a polynomial as the ratio of two polynomials.

`scipy.interpolate.`**`lagrange`**(*x*, *w*)
Return a Lagrange interpolating polynomial.

Given two 1-D arrays *x* and *w,* returns the Lagrange interpolating polynomial through the points `(x, w)`.

Warning: This implementation is numerically unstable. Do not expect to be able to use more than about 20

points even if they are chosen optimally.

Parameters
- **x** : array_like
  x represents the x-coordinates of a set of datapoints.
- **w** : array_like
  w represents the y-coordinates of a set of datapoints, i.e. f(*x*).

Returns
- **lagrange** : numpy.poly1d instance
  The Lagrange interpolating polynomial.

`scipy.interpolate.`**`approximate_taylor_polynomial`**(*f*, *x*, *degree*, *scale*, *order=None*)

Estimate the Taylor polynomial of f at x by polynomial fitting.

Parameters
- **f** : callable
  The function whose Taylor polynomial is sought. Should accept a vector of *x* values.
- **x** : scalar
  The point at which the polynomial is to be evaluated.
- **degree** : int
  The degree of the Taylor polynomial
- **scale** : scalar
  The width of the interval to use to evaluate the Taylor polynomial. Function values spread over a range this wide are used to fit the polynomial. Must be chosen carefully.
- **order** : int or None, optional
  The order of the polynomial to be used in the fitting; *f* will be evaluated `order+1` times. If None, use *degree*.

Returns
- **p** : poly1d instance
  The Taylor polynomial (translated to the origin, so that for example p(0)=f(x)).

Notes

The appropriate choice of “scale” is a trade-off; too large and the function differs from its Taylor polynomial too much to get a good answer, too small and round-off errors overwhelm the higher-order terms. The algorithm used becomes numerically unstable around order 30 even under ideal circumstances.

Choosing order somewhat larger than degree may improve the higher-order terms.

`scipy.interpolate.`**`pade`**(*an*, *m*)

Return Pade approximation to a polynomial as the ratio of two polynomials.

Parameters
- **an** : (N,) array_like
  Taylor series coefficients.
- **m** : int
  The order of the returned approximating polynomials.

Returns
- **p, q** : Polynomial class
  The Pade approximation of the polynomial defined by *an* is `p(x)/q(x)`.

Examples

```
>>> from scipy.interpolate import pade
>>> e_exp = [1.0, 1.0, 1.0/2.0, 1.0/6.0, 1.0/24.0, 1.0/120.0]
>>> p, q = pade(e_exp, 2)
```

```
>>> e_exp.reverse()
>>> e_poly = np.poly1d(e_exp)
```

Compare `e_poly(x)` and the Pade approximation `p(x)/q(x)`

```
>>> e_poly(1)
2.7166666666666668
```

```
>>> p(1)/q(1)
2.7179487179487181
```

See also:

scipy.ndimage.map_coordinates, *scipy.ndimage.spline_filter*, *scipy.signal.resample*, *scipy.signal.bspline*, *scipy.signal.gauss_spline*, *scipy.signal.qspline1d*, *scipy.signal.cspline1d*, *scipy.signal.qspline1d_eval*, *scipy.signal.cspline1d_eval*, *scipy.signal.qspline2d*, *scipy.signal.cspline2d*.

Functions existing for backward compatibility (should not be used in new code):

ppform(coeffs, breaks[, fill, sort])	Deprecated piecewise polynomial class.
spleval(*args, **kwds)	*spleval* is deprecated!
spline(*args, **kwds)	*spline* is deprecated!
splmake(*args, **kwds)	*splmake* is deprecated!
spltopp(*args, **kwds)	*spltopp* is deprecated!
pchip	alias of *PchipInterpolator*

class scipy.interpolate.**ppform**(*coeffs*, *breaks*, *fill=0.0*, *sort=False*)

Deprecated piecewise polynomial class.

New code should use the *PPoly* class instead.

Methods

__call__(x)	
antiderivative([nu])	Construct a new piecewise polynomial representing the antiderivative.
construct_fast(c, x[, extrapolate, axis])	Construct the piecewise polynomial without making checks.
derivative([nu])	Construct a new piecewise polynomial representing the derivative.
extend(c, x[, right])	Add additional breakpoints and coefficients to the polynomial.
from_bernstein_basis(bp[, extrapolate])	Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.
from_spline(tck[, extrapolate])	Construct a piecewise polynomial from a spline
fromspline(xk, cvals, order[, fill])	
integrate(a, b[, extrapolate])	Compute a definite integral over a piecewise polynomial.
roots([discontinuity, extrapolate])	Find real roots of the the piecewise polynomial.
solve([y, discontinuity, extrapolate])	Find real solutions of the the equation `pp(x) == y`.

ppform.**__call__**(*x*)

ppform.**antiderivative**(*nu=1*)

Construct a new piecewise polynomial representing the antiderivative.

Antiderivative is also the indefinite integral of the function, and derivative is its inverse operation.

Parameters **nu** : int, optional

Order of antiderivative to evaluate. Default is 1, i.e. compute the first integral. If negative, the derivative is returned.

Returns **pp** : PPoly
Piecewise polynomial of order k2 = k + n representing the antiderivative of this polynomial.

Notes

The antiderivative returned by this function is continuous and continuously differentiable to order n-1, up to floating point rounding error.

If antiderivative is computed and `self.extrapolate='periodic'`, it will be set to False for the returned instance. This is done because the antiderivative is no longer periodic and its correct evaluation outside of the initially given x interval is difficult.

`ppform.`**`construct_fast`**(*c*, *x*, *extrapolate=None*, *axis=0*)
Construct the piecewise polynomial without making checks.

Takes the same parameters as the constructor. Input arguments `c` and `x` must be arrays of the correct shape and type. The `c` array can only be of dtypes float and complex, and `x` array must have dtype float.

`ppform.`**`derivative`**(*nu=1*)
Construct a new piecewise polynomial representing the derivative.

Parameters **nu** : int, optional
Order of derivative to evaluate. Default is 1, i.e. compute the first derivative. If negative, the antiderivative is returned.

Returns **pp** : PPoly
Piecewise polynomial of order k2 = k - n representing the derivative of this polynomial.

Notes

Derivatives are evaluated piecewise for each polynomial segment, even if the polynomial is not differentiable at the breakpoints. The polynomial intervals are considered half-open, `[a, b)`, except for the last interval which is closed `[a, b]`.

`ppform.`**`extend`**(*c*, *x*, *right=None*)
Add additional breakpoints and coefficients to the polynomial.

Parameters **c** : ndarray, size (k, m, ...)
Additional coefficients for polynomials in intervals. Note that the first additional interval will be formed using one of the *self.x* end points.

x : ndarray, size (m,)
Additional breakpoints. Must be sorted in the same order as *self.x* and either to the right or to the left of the current breakpoints.

right
Deprecated argument. Has no effect.
Deprecated since version 0.19.

`ppform.`**`from_bernstein_basis`**(*bp*, *extrapolate=None*)
Construct a piecewise polynomial in the power basis from a polynomial in Bernstein basis.

Parameters **bp** : BPoly
A Bernstein basis polynomial, as created by BPoly

extrapolate : bool or 'periodic', optional
If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

`ppform.`**`from_spline`**(*tck*, *extrapolate=None*)
Construct a piecewise polynomial from a spline

Parameters **tck**

A spline, as returned by `splrep` or a BSpline object.

extrapolate : bool or 'periodic', optional

If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. Default is True.

classmethod `ppform.`**`fromspline`**(*xk*, *cvals*, *order*, *fill=0.0*)

`ppform.`**`integrate`**(*a*, *b*, *extrapolate=None*)

Compute a definite integral over a piecewise polynomial.

Parameters **a** : float

Lower integration bound

b : float

Upper integration bound

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to extrapolate to out-of-bounds points based on first and last intervals, or to return NaNs. If 'periodic', periodic extrapolation is used. If None (default), use *self.extrapolate*.

Returns **ig** : array_like

Definite integral of the piecewise polynomial over [a, b]

`ppform.`**`roots`**(*discontinuity=True*, *extrapolate=None*)

Find real roots of the the piecewise polynomial.

Parameters **discontinuity** : bool, optional

Whether to report sign changes across discontinuities at breakpoints as roots.

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns **roots** : ndarray

Roots of the polynomial(s).

If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

See also:

`PPoly.solve`

`ppform.`**`solve`**(*y=0.0*, *discontinuity=True*, *extrapolate=None*)

Find real solutions of the the equation `pp(x) == y`.

Parameters **y** : float, optional

Right-hand side. Default is zero.

discontinuity : bool, optional

Whether to report sign changes across discontinuities at breakpoints as roots.

extrapolate : {bool, 'periodic', None}, optional

If bool, determines whether to return roots from the polynomial extrapolated based on first and last intervals, 'periodic' works the same as False. If None (default), use *self.extrapolate*.

Returns **roots** : ndarray

Roots of the polynomial(s).

If the PPoly object describes multiple polynomials, the return value is an object array whose each element is an ndarray containing the roots.

Notes

This routine works only on real-valued polynomials.

If the piecewise polynomial contains sections that are identically zero, the root list will contain the start point of the corresponding interval, followed by a `nan` value.

If the polynomial is discontinuous across a breakpoint, and there is a sign change across the breakpoint, this is reported if the *discont* parameter is True.

Examples

Finding roots of `[x**2 - 1, (x - 1)**2]` defined on intervals `[-2, 1], [1, 2]`:

```
>>> from scipy.interpolate import PPoly
>>> pp = PPoly(np.array([[1, -4, 3], [1, 0, 0]]).T, [-2, 1, 2])
>>> pp.roots()
array([-1.,  1.])
```

`scipy.interpolate.`**`spleval`**`(*args, **kwds)`

`spleval` is deprecated! spleval is deprecated in scipy 0.19.0, use BSpline instead.

Evaluate a fixed spline represented by the given tuple at the new x-values
The *xj* values are the interior knot points. The approximation region is *xj[0]* to *xj[-1]*. If N+1 is the length of *xj*, then *cvals* should have length N+k where *k* is the order of the spline.

Parameters **(xj, cvals, k)** : tuple

Parameters that define the fixed spline

xj	[array_like] Interior knot points
cvals	[array_like] Curvature
k	[int] Order of the spline
xnew	[array_like] Locations to calculate spline
deriv	[int] Deriv

Returns **spleval** : ndarray

If *cvals* represents more than one curve (*cvals.ndim* > 1) and/or *xnew* is N-d, then the result is *xnew.shape* + *cvals.shape[1:]* providing the interpolation of multiple curves.

Notes

Internally, an additional *k*-1 knot points are added on either side of the spline.

`scipy.interpolate.`**`spline`**`(*args, **kwds)`

`spline` is deprecated! spline is deprecated in scipy 0.19.0, use Bspline class instead.

Interpolate a curve at new points using a spline fit

Parameters **xk, yk** : array_like

The x and y values that define the curve.

xnew	[array_like] The x values where spline should estimate the y values.
order	[int] Default is 3.
kind	[string] One of {'smoothest'}
conds	[Don't know] Don't know

Returns **spline** : ndarray

An array of y values; the spline evaluated at the positions *xnew*.

`scipy.interpolate.`**`splmake`**`(*args, **kwds)`

`splmake` is deprecated! splmake is deprecated in scipy 0.19.0, use make_interp_spline instead.

Return a representation of a spline given data-points at internal knots

Parameters **xk** : array_like
The input array of x values of rank 1

yk	[array_like] The input array of y values of rank N. *yk* can be an N-d array to represent more than one curve, through the same *xk* points. The first dimension is assumed to be the interpolating dimension and is the same length of *xk*.
order	[int, optional] Order of the spline
kind	[str, optional] Can be 'smoothest', 'not_a_knot', 'fixed', 'clamped', 'natural', 'periodic', 'symmetric', 'user', 'mixed' and it is ignored if order < 2
conds	[optional] Conds

Returns **splmake** : tuple
Return a (*xk*, *cvals*, *k*) representation of a spline given data-points where the (internal) knots are at the data-points.

`scipy.interpolate.`**`spltopp`**(**args*, ***kwds*)
spltopp is deprecated! spltopp is deprecated in scipy 0.19.0, use PPoly.from_spline instead.

Return a piece-wise polynomial object from a fixed-spline tuple.

`scipy.interpolate.`**`pchip`**
alias of *PchipInterpolator*

5.8 Input and output (`scipy.io`)

SciPy has many modules, classes, and functions available to read data from and write data to a variety of file formats.

See also:

numpy-reference.routines.io (in Numpy)

5.8.1 MATLAB® files

loadmat(file_name[, mdict, appendmat])	Load MATLAB file.
savemat(file_name, mdict[, appendmat, ...])	Save a dictionary of names and arrays into a MATLAB-style .mat file.
whosmat(file_name[, appendmat])	List variables inside a MATLAB file.

`scipy.io.`**`loadmat`**(*file_name*, *mdict=None*, *appendmat=True*, ***kwargs*)
Load MATLAB file.

Parameters **file_name** : str
Name of the mat file (do not need .mat extension if appendmat==True). Can also pass open file-like object.
mdict : dict, optional
Dictionary in which to insert matfile variables.
appendmat : bool, optional
True to append the .mat extension to the end of the given filename, if not already present.

byte_order : str or None, optional
None by default, implying byte order guessed from mat file. Otherwise can be one of ('native', '=', 'little', '<', 'BIG', '>').

mat_dtype : bool, optional
If True, return arrays in same dtype as would be loaded into MATLAB (instead of the dtype with which they are saved).

squeeze_me : bool, optional
Whether to squeeze unit matrix dimensions or not.

chars_as_strings : bool, optional
Whether to convert char arrays to string arrays.

matlab_compatible : bool, optional
Returns matrices as would be loaded by MATLAB (implies squeeze_me=False, chars_as_strings=False, mat_dtype=True, struct_as_record=True).

struct_as_record : bool, optional
Whether to load MATLAB structs as numpy record arrays, or as old-style numpy arrays with dtype=object. Setting this flag to False replicates the behavior of scipy version 0.7.x (returning numpy object arrays). The default setting is True, because it allows easier round-trip load and save of MATLAB files.

verify_compressed_data_integrity : bool, optional
Whether the length of compressed sequences in the MATLAB file should be checked, to ensure that they are not longer than we expect. It is advisable to enable this (the default) because overlong compressed sequences in MATLAB files generally indicate that the files have experienced some sort of corruption.

variable_names : None or sequence
If None (the default) - read all variables in file. Otherwise *variable_names* should be a sequence of strings, giving names of the matlab variables to read from the file. The reader will skip any variable with a name not in this sequence, possibly saving some read processing.

Returns

mat_dict : dict
dictionary with variable names as keys, and loaded matrices as values.

Notes

v4 (Level 1.0), v6 and v7 to 7.2 matfiles are supported.

You will need an HDF5 python library to read matlab 7.3 format mat files. Because scipy does not supply one, we do not implement the HDF5 / 7.3 interface here.

`scipy.io.`**`savemat`**(*file_name*, *mdict*, *appendmat=True*, *format='5'*, *long_field_names=False*, *do_compression=False*, *oned_as='row'*)

Save a dictionary of names and arrays into a MATLAB-style .mat file.

This saves the array objects in the given dictionary to a MATLAB- style .mat file.

Parameters

file_name : str or file-like object
Name of the .mat file (.mat extension not needed if `appendmat == True`). Can also pass open file_like object.

mdict : dict
Dictionary from which to save matfile variables.

appendmat : bool, optional
True (the default) to append the .mat extension to the end of the given filename, if not already present.

format : {'5', '4'}, string, optional
'5' (the default) for MATLAB 5 and up (to 7.2), '4' for MATLAB 4 .mat files.

long_field_names : bool, optional

False (the default) - maximum field name length in a structure is 31 characters which is the documented maximum length. True - maximum field name length in a structure is 63 characters which works for MATLAB 7.6+.

do_compression : bool, optional
Whether or not to compress matrices on write. Default is False.

oned_as : {'row', 'column'}, optional
If 'column', write 1-D numpy arrays as column vectors. If 'row', write 1-D numpy arrays as row vectors.

See also:

`mio4.MatFile4Writer`, `mio5.MatFile5Writer`

`scipy.io.whosmat` (*file_name*, *appendmat=True*, ***kwargs*)
List variables inside a MATLAB file.

Parameters

file_name : str
Name of the mat file (do not need .mat extension if appendmat==True) Can also pass open file-like object.

appendmat : bool, optional
True to append the .mat extension to the end of the given filename, if not already present.

byte_order : str or None, optional
None by default, implying byte order guessed from mat file. Otherwise can be one of ('native', '=', 'little', '<', 'BIG', '>').

mat_dtype : bool, optional
If True, return arrays in same dtype as would be loaded into MATLAB (instead of the dtype with which they are saved).

squeeze_me : bool, optional
Whether to squeeze unit matrix dimensions or not.

chars_as_strings : bool, optional
Whether to convert char arrays to string arrays.

matlab_compatible : bool, optional
Returns matrices as would be loaded by MATLAB (implies squeeze_me=False, chars_as_strings=False, mat_dtype=True, struct_as_record=True).

struct_as_record : bool, optional
Whether to load MATLAB structs as numpy record arrays, or as old-style numpy arrays with dtype=object. Setting this flag to False replicates the behavior of scipy version 0.7.x (returning numpy object arrays). The default setting is True, because it allows easier round-trip load and save of MATLAB files.

Returns

variables : list of tuples
A list of tuples, where each tuple holds the matrix name (a string), its shape (tuple of ints), and its data class (a string). Possible data classes are: int8, uint8, int16, uint16, int32, uint32, int64, uint64, single, double, cell, struct, object, char, sparse, function, opaque, logical, unknown.

Notes

v4 (Level 1.0), v6 and v7 to 7.2 matfiles are supported.

You will need an HDF5 python library to read matlab 7.3 format mat files. Because scipy does not supply one, we do not implement the HDF5 / 7.3 interface here.

New in version 0.12.0.

5.8.2 IDL® files

readsav(file_name[, idict, python_dict, ...])	Read an IDL .sav file.

`scipy.io.readsav`(*file_name*, *idict=None*, *python_dict=False*, *uncompressed_file_name=None*, *verbose=False*)

Read an IDL .sav file.

Parameters

file_name : str
Name of the IDL save file.

idict : dict, optional
Dictionary in which to insert .sav file variables.

python_dict : bool, optional
By default, the object return is not a Python dictionary, but a case-insensitive dictionary with item, attribute, and call access to variables. To get a standard Python dictionary, set this option to True.

uncompressed_file_name : str, optional
This option only has an effect for .sav files written with the /compress option. If a file name is specified, compressed .sav files are uncompressed to this file. Otherwise, readsav will use the `tempfile` module to determine a temporary filename automatically, and will remove the temporary file upon successfully reading it in.

verbose : bool, optional
Whether to print out information about the save file, including the records read, and available variables.

Returns

idl_dict : AttrDict or dict
If *python_dict* is set to False (default), this function returns a case-insensitive dictionary with item, attribute, and call access to variables. If *python_dict* is set to True, this function returns a Python dictionary with all variable names in lowercase. If *idict* was specified, then variables are written to the dictionary specified, and the updated dictionary is returned.

5.8.3 Matrix Market files

mminfo(source)	Return size and storage parameters from Matrix Market file-like 'source'.
mmread(source)	Reads the contents of a Matrix Market file-like 'source' into a matrix.
mmwrite(target, a[, comment, field, ...])	Writes the sparse or dense array *a* to Matrix Market file-like *target*.

`scipy.io.mminfo`(*source*)

Return size and storage parameters from Matrix Market file-like 'source'.

Parameters

source : str or file-like
Matrix Market filename (extension .mtx) or open file-like object

Returns

rows : int
Number of matrix rows.

cols : int
Number of matrix columns.

entries : int
Number of non-zero entries of a sparse matrix or rows*cols for a dense matrix.

format : str
Either 'coordinate' or 'array'.

field : str
Either 'real', 'complex', 'pattern', or 'integer'.
symmetry : str
Either 'general', 'symmetric', 'skew-symmetric', or 'hermitian'.

`scipy.io.mmread(source)`
Reads the contents of a Matrix Market file-like 'source' into a matrix.

Parameters **source** : str or file-like
Matrix Market filename (extensions .mtx, .mtz.gz) or open file-like object.

Returns **a** : ndarray or coo_matrix
Dense or sparse matrix depending on the matrix format in the Matrix Market file.

`scipy.io.mmwrite(target, a, comment='', field=None, precision=None, symmetry=None)`
Writes the sparse or dense array *a* to Matrix Market file-like *target*.

Parameters **target** : str or file-like
Matrix Market filename (extension .mtx) or open file-like object.
a : array like
Sparse or dense 2D array.
comment : str, optional
Comments to be prepended to the Matrix Market file.
field : None or str, optional
Either 'real', 'complex', 'pattern', or 'integer'.
precision : None or int, optional
Number of digits to display for real or complex values.
symmetry : None or str, optional
Either 'general', 'symmetric', 'skew-symmetric', or 'hermitian'. If symmetry is None the symmetry type of 'a' is determined by its values.

5.8.4 Unformatted Fortran files

FortranFile(filename[, mode, header_dtype])	A file object for unformatted sequential files from Fortran code.

class `scipy.io.FortranFile(filename, mode='r', header_dtype=<type 'numpy.uint32'>)`
A file object for unformatted sequential files from Fortran code.

Parameters **filename** : file or str
Open file object or filename.
mode : {'r', 'w'}, optional
Read-write mode, default is 'r'.
header_dtype : dtype, optional
Data type of the header. Size and endiness must match the input/output file.

Notes

These files are broken up into records of unspecified types. The size of each record is given at the start (although the size of this header is not standard) and the data is written onto disk without any formatting. Fortran compilers supporting the BACKSPACE statement will write a second copy of the size to facilitate backwards seeking.

This class only supports files written with both sizes for the record. It also does not support the subrecords used in Intel and gfortran compilers for records which are greater than 2GB with a 4-byte header.

An example of an unformatted sequential file in Fortran would be written as:

```
OPEN(1, FILE=myfilename, FORM='unformatted')

WRITE(1) myvariable
```

Since this is a non-standard file format, whose contents depend on the compiler and the endianness of the machine, caution is advised. Files from gfortran 4.8.0 and gfortran 4.1.2 on x86_64 are known to work.

Consider using Fortran direct-access files or files from the newer Stream I/O, which can be easily read by `numpy.fromfile`.

Examples

To create an unformatted sequential Fortran file:

```
>>> from scipy.io import FortranFile
>>> f = FortranFile('test.unf', 'w')
>>> f.write_record(np.array([1,2,3,4,5], dtype=np.int32))
>>> f.write_record(np.linspace(0,1,20).reshape((5,-1)))
>>> f.close()
```

To read this file:

```
>>> from scipy.io import FortranFile
>>> f = FortranFile('test.unf', 'r')
>>> print(f.read_ints(dtype=np.int32))
[1 2 3 4 5]
>>> print(f.read_reals(dtype=float).reshape((5,-1)))
[[ 0.          0.05263158  0.10526316  0.15789474]
 [ 0.21052632  0.26315789  0.31578947  0.36842105]
 [ 0.42105263  0.47368421  0.52631579  0.57894737]
 [ 0.63157895  0.68421053  0.73684211  0.78947368]
 [ 0.84210526  0.89473684  0.94736842  1.        ]]
>>> f.close()
```

Methods

`close`()	Closes the file.
`read_ints`([dtype])	Reads a record of a given type from the file, defaulting to an integer type (`INTEGER*4` in Fortran).
`read_reals`([dtype])	Reads a record of a given type from the file, defaulting to a floating point number (`real*8` in Fortran).
`read_record`([dtype])	Reads a record of a given type from the file.
`write_record`(s)	Write a record (including sizes) to the file.

`FortranFile.`**`close`**`()`

Closes the file. It is unsupported to call any other methods off this object after closing it. Note that this class supports the 'with' statement in modern versions of Python, to call this automatically

`FortranFile.`**`read_ints`**`(`*dtype='i4'*`)`

Reads a record of a given type from the file, defaulting to an integer type (`INTEGER*4` in Fortran).

Parameters **dtype** : dtype, optional
Data type specifying the size and endiness of the data.

Returns **data** : ndarray
A one-dimensional array object.

See also:

read_reals, *read_record*

FortranFile.**read_reals**(*dtype='f8'*)

Reads a record of a given type from the file, defaulting to a floating point number (real*8 in Fortran).

Parameters **dtype** : dtype, optional
Data type specifying the size and endiness of the data.

Returns **data** : ndarray
A one-dimensional array object.

See also:

read_ints, *read_record*

FortranFile.**read_record**(*dtype=None*)

Reads a record of a given type from the file.

Parameters **dtype** : dtype, optional
Data type specifying the size and endiness of the data.

Returns **data** : ndarray
A one-dimensional array object.

See also:

read_reals, *read_ints*

Notes

If the record contains a multi-dimensional array, calling reshape or resize will restructure the array to the correct size. Since Fortran multidimensional arrays are stored in column-major format, this may have some non-intuitive consequences. If the variable was declared as 'INTEGER var(5,4)', for example, var could be read with 'read_record(dtype=np.integer).reshape((4,5))' since Python uses row-major ordering of indices.

One can transpose to obtain the indices in the same order as in Fortran.

For records that contain several variables or mixed types (as opposed to single scalar or array types), it is possible to specify a dtype with mixed types:

```
record = f.read_record([('a', '<f4'), ('b', '<i4')])
record['a']  # access the variable 'a'
```

and if any of the variables are arrays, the shape can be specified as the third item in the relevant tuple:

```
record = f.read_record([('a', '<f4'), ('b', '<i4', (3,3))])
```

Numpy also supports a short syntax for this kind of type:

```
record = f.read_record('<f4,(3,3)<i4')
record['f0']  # variables are called f0, f1, ...
```

FortranFile.**write_record**(*s*)

Write a record (including sizes) to the file.

Parameters **s** : array_like
The data to write.

5.8.5 Netcdf

netcdf_file(filename[, mode, mmap, version, ...])	A file object for NetCDF data.
netcdf_variable(data, typecode, size, shape, ...)	A data object for the *netcdf* module.

class scipy.io.**netcdf_file** (*filename*, *mode='r'*, *mmap=None*, *version=1*, *maskandscale=False*)

A file object for NetCDF data.

A *netcdf_file* object has two standard attributes: *dimensions* and *variables*. The values of both are dictionaries, mapping dimension names to their associated lengths and variable names to variables, respectively. Application programs should never modify these dictionaries.

All other attributes correspond to global attributes defined in the NetCDF file. Global file attributes are created by assigning to an attribute of the *netcdf_file* object.

Parameters

filename : string or file-like
string -> filename

mode : {'r', 'w', 'a'}, optional
read-write-append mode, default is 'r'

mmap : None or bool, optional
Whether to mmap *filename* when reading. Default is True when *filename* is a file name, False when *filename* is a file-like object. Note that when mmap is in use, data arrays returned refer directly to the mmapped data on disk, and the file cannot be closed as long as references to it exist.

version : {1, 2}, optional
version of netcdf to read / write, where 1 means *Classic format* and 2 means *64-bit offset format*. Default is 1. See here for more info.

maskandscale : bool, optional
Whether to automatically scale and/or mask data based on attributes. Default is False.

Notes

The major advantage of this module over other modules is that it doesn't require the code to be linked to the NetCDF libraries. This module is derived from pupynere.

NetCDF files are a self-describing binary data format. The file contains metadata that describes the dimensions and variables in the file. More details about NetCDF files can be found here. There are three main sections to a NetCDF data structure:

1. Dimensions
2. Variables
3. Attributes

The dimensions section records the name and length of each dimension used by the variables. The variables would then indicate which dimensions it uses and any attributes such as data units, along with containing the data values for the variable. It is good practice to include a variable that is the same name as a dimension to provide the values for that axes. Lastly, the attributes section would contain additional information such as the name of the file creator or the instrument used to collect the data.

When writing data to a NetCDF file, there is often the need to indicate the 'record dimension'. A record dimension is the unbounded dimension for a variable. For example, a temperature variable may have dimensions of latitude, longitude and time. If one wants to add more temperature data to the NetCDF file as time progresses, then the temperature variable should have the time dimension flagged as the record dimension.

In addition, the NetCDF file header contains the position of the data in the file, so access can be done in an efficient manner without loading unnecessary data into memory. It uses the `mmap` module to create Numpy arrays mapped to the data on disk, for the same purpose.

Note that when `netcdf_file` is used to open a file with mmap=True (default for read-only), arrays returned by it refer to data directly on the disk. The file should not be closed, and cannot be cleanly closed when asked, if such arrays are alive. You may want to copy data arrays obtained from mmapped Netcdf file if they are to be processed after the file is closed, see the example below.

Examples

To create a NetCDF file:

```
>>> from scipy.io import netcdf
>>> f = netcdf.netcdf_file('simple.nc', 'w')
>>> f.history = 'Created for a test'
>>> f.createDimension('time', 10)
>>> time = f.createVariable('time', 'i', ('time',))
>>> time[:] = np.arange(10)
>>> time.units = 'days since 2008-01-01'
>>> f.close()
```

Note the assignment of `range(10)` to `time[:]`. Exposing the slice of the time variable allows for the data to be set in the object, rather than letting `range(10)` overwrite the `time` variable.

To read the NetCDF file we just created:

```
>>> from scipy.io import netcdf
>>> f = netcdf.netcdf_file('simple.nc', 'r')
>>> print(f.history)
Created for a test
>>> time = f.variables['time']
>>> print(time.units)
days since 2008-01-01
>>> print(time.shape)
(10,)
>>> print(time[-1])
9
```

NetCDF files, when opened read-only, return arrays that refer directly to memory-mapped data on disk:

```
>>> data = time[:]
>>> data.base.base
<mmap.mmap object at 0x7fe753763180>
```

If the data is to be processed after the file is closed, it needs to be copied to main memory:

```
>>> data = time[:].copy()
>>> f.close()
>>> data.mean()
4.5
```

A NetCDF file can also be used as context manager:

```
>>> from scipy.io import netcdf
>>> with netcdf.netcdf_file('simple.nc', 'r') as f:
...     print(f.history)
Created for a test
```

Methods

close()	Closes the NetCDF file.
createDimension(name, length)	Adds a dimension to the Dimension section of the NetCDF data structure.
createVariable(name, type, dimensions)	Create an empty variable for the *netcdf_file* object, specifying its data type and the dimensions it uses.
flush()	Perform a sync-to-disk flush if the *netcdf_file* object is in write mode.
sync()	Perform a sync-to-disk flush if the *netcdf_file* object is in write mode.

netcdf_file.**close**()

Closes the NetCDF file.

netcdf_file.**createDimension**(*name*, *length*)

Adds a dimension to the Dimension section of the NetCDF data structure.

Note that this function merely adds a new dimension that the variables can reference. The values for the dimension, if desired, should be added as a variable using *createVariable*, referring to this dimension.

Parameters

name : str
Name of the dimension (Eg, 'lat' or 'time').

length : int
Length of the dimension.

See also:

createVariable

netcdf_file.**createVariable**(*name*, *type*, *dimensions*)

Create an empty variable for the *netcdf_file* object, specifying its data type and the dimensions it uses.

Parameters

name : str
Name of the new variable.

type : dtype or str
Data type of the variable.

dimensions : sequence of str
List of the dimension names used by the variable, in the desired order.

Returns

variable : netcdf_variable
The newly created netcdf_variable object. This object has also been added to the *netcdf_file* object as well.

See also:

createDimension

Notes

Any dimensions to be used by the variable should already exist in the NetCDF data structure or should be created by *createDimension* prior to creating the NetCDF variable.

netcdf_file.**flush**()

Perform a sync-to-disk flush if the *netcdf_file* object is in write mode.

See also:

sync	Identical function

netcdf_file.**sync**()
Perform a sync-to-disk flush if the *netcdf_file* object is in write mode.

See also:

sync	Identical function

class scipy.io.**netcdf_variable**(*data*, *typecode*, *size*, *shape*, *dimensions*, *attributes=None*, *maskandscale=False*)
A data object for the *netcdf* module.

netcdf_variable objects are constructed by calling the method *netcdf_file.createVariable* on the *netcdf_file* object. *netcdf_variable* objects behave much like array objects defined in numpy, except that their data resides in a file. Data is read by indexing and written by assigning to an indexed subset; the entire array can be accessed by the index [:] or (for scalars) by using the methods *getValue* and *assignValue*. *netcdf_variable* objects also have attribute shape with the same meaning as for arrays, but the shape cannot be modified. There is another read-only attribute *dimensions*, whose value is the tuple of dimension names.

All other attributes correspond to variable attributes defined in the NetCDF file. Variable attributes are created by assigning to an attribute of the *netcdf_variable* object.

Parameters

data : array_like
The data array that holds the values for the variable. Typically, this is initialized as empty, but with the proper shape.

typecode : dtype character code
Desired data-type for the data array.

size : int
Desired element size for the data array.

shape : sequence of ints
The shape of the array. This should match the lengths of the variable's dimensions.

dimensions : sequence of strings
The names of the dimensions used by the variable. Must be in the same order of the dimension lengths given by shape.

attributes : dict, optional
Attribute values (any type) keyed by string names. These attributes become attributes for the netcdf_variable object.

maskandscale : bool, optional
Whether to automatically scale and/or mask data based on attributes. Default is False.

See also:

isrec, shape

Attributes

dimensions	(list of str) List of names of dimensions used by the variable object.
isrec, shape	Properties

Methods

assignValue(value)	Assign a scalar value to a *netcdf_variable* of length one.
getValue()	Retrieve a scalar value from a *netcdf_variable* of length one.
itemsize()	Return the itemsize of the variable.

Continued on next page

Table 5.68 – continued from previous page

typecode()	Return the typecode of the variable.

netcdf_variable.**assignValue**(*value*)
Assign a scalar value to a *netcdf_variable* of length one.

Parameters **value** : scalar
Scalar value (of compatible type) to assign to a length-one netcdf variable. This value will be written to file.

Raises **ValueError**
If the input is not a scalar, or if the destination is not a length-one netcdf variable.

netcdf_variable.**getValue**()
Retrieve a scalar value from a *netcdf_variable* of length one.

Raises **ValueError**
If the netcdf variable is an array of length greater than one, this exception will be raised.

netcdf_variable.**itemsize**()
Return the itemsize of the variable.

Returns **itemsize** : int
The element size of the variable (eg, 8 for float64).

netcdf_variable.**typecode**()
Return the typecode of the variable.

Returns **typecode** : char
The character typecode of the variable (eg, 'i' for int).

5.8.6 Harwell-Boeing files

hb_read(file)	Read HB-format file.
hb_write(file, m[, hb_info])	Write HB-format file.

scipy.io.**hb_read**(*file*)
Read HB-format file.

Parameters **file** : str-like or file-like
If a string-like object, file is the name of the file to read. If a file-like object, the data are read from it.

Returns **data** : scipy.sparse.csc_matrix instance
The data read from the HB file as a sparse matrix.

Notes

At the moment not the full Harwell-Boeing format is supported. Supported features are:

- assembled, non-symmetric, real matrices
- integer for pointer/indices
- exponential format for float values, and int format

scipy.io.**hb_write**(*file*, *m*, *hb_info=None*)
Write HB-format file.

Parameters

file : str-like or file-like
: if a string-like object, file is the name of the file to read. If a file-like object, the data are read from it.

m : sparse-matrix
: the sparse matrix to write

hb_info : HBInfo
: contains the meta-data for write

Returns

None

Notes

At the moment not the full Harwell-Boeing format is supported. Supported features are:

- assembled, non-symmetric, real matrices
- integer for pointer/indices
- exponential format for float values, and int format

5.8.7 Wav sound files (`scipy.io.wavfile`)

read(filename[, mmap])	Open a WAV file
write(filename, rate, data)	Write a numpy array as a WAV file.
WavFileWarning	

`scipy.io.wavfile.`**`read`**(*filename*, *mmap=False*)

Open a WAV file

Return the sample rate (in samples/sec) and data from a WAV file.

Parameters

filename : string or open file handle
: Input wav file.

mmap : bool, optional
: Whether to read data as memory-mapped. Only to be used on real files (Default: False). New in version 0.12.0.

Returns

rate : int
: Sample rate of wav file.

data : numpy array
: Data read from wav file. Data-type is determined from the file; see Notes.

Notes

This function cannot read wav files with 24-bit data.

Common data types: *[R93]*

WAV format	Min	Max	NumPy dtype
32-bit floating-point	-1.0	+1.0	float32
32-bit PCM	-2147483648	+2147483647	int32
16-bit PCM	-32768	+32767	int16
8-bit PCM	0	255	uint8

Note that 8-bit PCM is unsigned.

References

[R93]

`scipy.io.wavfile.write`(*filename*, *rate*, *data*)

Write a numpy array as a WAV file.

Parameters

filename : string or open file handle
: Output wav file.

rate : int
: The sample rate (in samples/sec).

data : ndarray
: A 1-D or 2-D numpy array of either integer or float data-type.

Notes

- Writes a simple uncompressed WAV file.
- To write multiple-channels, use a 2-D array of shape (Nsamples, Nchannels).
- The bits-per-sample and PCM/float will be determined by the data-type.

Common data types: *[R94]*

WAV format	Min	Max	NumPy dtype
32-bit floating-point	-1.0	+1.0	float32
32-bit PCM	-2147483648	+2147483647	int32
16-bit PCM	-32768	+32767	int16
8-bit PCM	0	255	uint8

Note that 8-bit PCM is unsigned.

References

[R94]

exception `scipy.io.wavfile.WavFileWarning`

5.8.8 Arff files (`scipy.io.arff`)

loadarff(f)	Read an arff file.
MetaData(rel, attr)	Small container to keep useful informations on a ARFF dataset.
ArffError	
ParseArffError	

`scipy.io.arff.loadarff`(*f*)

Read an arff file.

The data is returned as a record array, which can be accessed much like a dictionary of numpy arrays. For example, if one of the attributes is called 'pressure', then its first 10 data points can be accessed from the `data` record array like so: `data['pressure'][0:10]`

Parameters

f : file-like or str
: File-like object to read from, or filename to open.

Returns

data : record array
: The data of the arff file, accessible by attribute names.

meta : *MetaData*
: Contains information about the arff file such as name and type of attributes, the relation (name of the dataset), etc...

Raises **ParseArffError**
This is raised if the given file is not ARFF-formatted.

NotImplementedError
The ARFF file has an attribute which is not supported yet.

Notes

This function should be able to read most arff files. Not implemented functionality include:

- date type attributes
- string type attributes

It can read files with numeric and nominal attributes. It cannot read files with sparse data ({} in the file). However, this function can read files with missing data (? in the file), representing the data points as NaNs.

Examples

```
>>> from scipy.io import arff
>>> from cStringIO import StringIO
>>> content = """
... @relation foo
... @attribute width  numeric
... @attribute height numeric
... @attribute color  {red,green,blue,yellow,black}
... @data
... 5.0,3.25,blue
... 4.5,3.75,green
... 3.0,4.00,red
... """
>>> f = StringIO(content)
>>> data, meta = arff.loadarff(f)
>>> data
array([(5.0, 3.25, 'blue'), (4.5, 3.75, 'green'), (3.0, 4.0, 'red')],
      dtype=[('width', '<f8'), ('height', '<f8'), ('color', '|S6')])
>>> meta
Dataset: foo
    width's type is numeric
    height's type is numeric
    color's type is nominal, range is ('red', 'green', 'blue', 'yellow', 'black')
```

class `scipy.io.arff.MetaData` (*rel*, *attr*)
Small container to keep useful informations on a ARFF dataset.

Knows about attributes names and types.

Notes

Also maintains the list of attributes in order, i.e. doing for i in meta, where meta is an instance of MetaData, will return the different attribute names in the order they were defined.

Examples

```
data, meta = loadarff('iris.arff')
# This will print the attributes names of the iris.arff dataset
for i in meta:
    print i
# This works too
meta.names()
```

```
# Getting attribute type
types = meta.types()
```

Methods

names()	Return the list of attribute names.
types()	Return the list of attribute types.

MetaData.**names**()
Return the list of attribute names.

MetaData.**types**()
Return the list of attribute types.

exception scipy.io.arff.**ArffError**

exception scipy.io.arff.**ParseArffError**

5.9 Linear algebra (scipy.linalg)

Linear algebra functions.

See also:

numpy.linalg for more linear algebra functions. Note that although *scipy.linalg* imports most of them, identically named functions from *scipy.linalg* may offer more or slightly differing functionality.

5.9.1 Basics

inv(a[, overwrite_a, check_finite])	Compute the inverse of a matrix.
solve(a, b[, sym_pos, lower, overwrite_a, ...])	Solves the linear equation set a * x = b for the unknown x for square a matrix.
solve_banded(l_and_u, ab, b[, overwrite_ab, ...])	Solve the equation a x = b for x, assuming a is banded matrix.
solveh_banded(ab, b[, overwrite_ab, ...])	Solve equation a x = b.
solve_circulant(c, b[, singular, tol, ...])	Solve C x = b for x, where C is a circulant matrix.
solve_triangular(a, b[, trans, lower, ...])	Solve the equation *a x = b* for *x*, assuming a is a triangular matrix.
solve_toeplitz(c_or_cr, b[, check_finite])	Solve a Toeplitz system using Levinson Recursion
det(a[, overwrite_a, check_finite])	Compute the determinant of a matrix
norm(a[, ord, axis, keepdims])	Matrix or vector norm.
lstsq(a, b[, cond, overwrite_a, ...])	Compute least-squares solution to equation Ax = b.
pinv(a[, cond, rcond, return_rank, check_finite])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
pinv2(a[, cond, rcond, return_rank, ...])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
pinvh(a[, cond, rcond, lower, return_rank, ...])	Compute the (Moore-Penrose) pseudo-inverse of a Hermitian matrix.
kron(a, b)	Kronecker product.

Continued on next page

Table 5.73 – continued from previous page

tril(m[, k])	Make a copy of a matrix with elements above the k-th diagonal zeroed.
triu(m[, k])	Make a copy of a matrix with elements below the k-th diagonal zeroed.
orthogonal_procrustes(A, B[, check_finite])	Compute the matrix solution of the orthogonal Procrustes problem.
matrix_balance(A[, permute, scale, ...])	Compute a diagonal similarity transformation for row/column balancing.
LinAlgError	Generic Python-exception-derived object raised by linalg functions.

scipy.linalg.**inv**(*a*, *overwrite_a=False*, *check_finite=True*)

Compute the inverse of a matrix.

Parameters **a** : array_like
Square matrix to be inverted.
overwrite_a : bool, optional
Discard data in *a* (may improve performance). Default is False.
check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **ainv** : ndarray
Inverse of the matrix *a*.

Raises **LinAlgError**
If *a* is singular.
ValueError
If *a* is not square, or not 2-dimensional.

Examples

```
>>> from scipy import linalg
>>> a = np.array([[1., 2.], [3., 4.]])
>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.dot(a, linalg.inv(a))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

scipy.linalg.**solve**(*a*, *b*, *sym_pos=False*, *lower=False*, *overwrite_a=False*, *overwrite_b=False*, *debug=None*, *check_finite=True*, *assume_a='gen'*, *transposed=False*)

Solves the linear equation set `a * x = b` for the unknown `x` for square `a` matrix.

If the data matrix is known to be a particular type then supplying the corresponding string to `assume_a` key chooses the dedicated solver. The available options are

generic matrix	'gen'
symmetric	'sym'
hermitian	'her'
positive definite	'pos'

If omitted, `'gen'` is the default structure.

The datatype of the arrays define which solver is called regardless of the values. In other words, even when the complex array entries have precisely zero imaginary parts, the complex solver will be called based on the data type of the array.

Parameters

a : (N, N) array_like
    Square input data

b : (N, NRHS) array_like
    Input data for the right hand side.

sym_pos : bool, optional
    Assume *a* is symmetric and positive definite. This key is deprecated and assume_a = 'pos' keyword is recommended instead. The functionality is the same. It will be removed in the future.

lower : bool, optional
    If True, only the data contained in the lower triangle of *a*. Default is to use upper triangle. (ignored for `'gen'`)

overwrite_a : bool, optional
    Allow overwriting data in *a* (may enhance performance). Default is False.

overwrite_b : bool, optional
    Allow overwriting data in *b* (may enhance performance). Default is False.

check_finite : bool, optional
    Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

assume_a : str, optional
    Valid entries are explained above.

transposed: bool, optional
    If True, depending on the data type `a^T x = b` or `a^H x = b` is solved (only taken into account for `'gen'`).

Returns

x : (N, NRHS) ndarray
    The solution array.

Raises

ValueError
    If size mismatches detected or input a is not square.

LinAlgError
    If the matrix is singular.

RuntimeWarning
    If an ill-conditioned input a is detected.

Notes

If the input b matrix is a 1D array with N elements, when supplied together with an NxN input a, it is assumed as a valid column vector despite the apparent size mismatch. This is compatible with the numpy.dot() behavior and the returned result is still 1D array.

The generic, symmetric, hermitian and positive definite solutions are obtained via calling ?GESVX, ?SYSVX, ?HESVX, and ?POSVX routines of LAPACK respectively.

Examples

Given *a* and *b*, solve for *x*:

```
>>> a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
>>> b = np.array([2, 4, -1])
>>> from scipy import linalg
>>> x = linalg.solve(a, b)
>>> x
array([ 2., -2.,  9.])
```

```
>>> np.dot(a, x) == b
array([ True,  True,  True], dtype=bool)
```

scipy.linalg.**solve_banded**(*l_and_u*, *ab*, *b*, *overwrite_ab=False*, *overwrite_b=False*, *debug=None*, *check_finite=True*)

Solve the equation a x = b for x, assuming a is banded matrix.

The matrix a is stored in *ab* using the matrix diagonal ordered form:

```
ab[u + i - j, j] == a[i,j]
```

Example of *ab* (shape of a is (6,6), *u* =1, *l* =2):

```
*    a01  a12  a23  a34  a45
a00  a11  a22  a33  a44  a55
a10  a21  a32  a43  a54   *
a20  a31  a42  a53   *    *
```

Parameters

(l, u) : (integer, integer)
  Number of non-zero lower and upper diagonals

ab : ($l + u + 1$, M) array_like
  Banded matrix

b : (M,) or (M, K) array_like
  Right-hand side

overwrite_ab : bool, optional
  Discard data in *ab* (may enhance performance)

overwrite_b : bool, optional
  Discard data in *b* (may enhance performance)

check_finite : bool, optional
  Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

x : (M,) or (M, K) ndarray
  The solution to the system a x = b. Returned shape depends on the shape of *b*.

scipy.linalg.**solveh_banded**(*ab*, *b*, *overwrite_ab=False*, *overwrite_b=False*, *lower=False*, *check_finite=True*)

Solve equation a x = b. a is Hermitian positive-definite banded matrix.

The matrix a is stored in *ab* either in lower diagonal or upper diagonal ordered form:

ab[u + i - j, j] == a[i,j] (if upper form; i <= j) ab[i - j, j] == a[i,j] (if lower form; i >= j)

Example of *ab* (shape of a is (6, 6), *u* =2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters

ab : ($u + 1$, M) array_like

Banded matrix

b : (M,) or (M, K) array_like
Right-hand side

overwrite_ab : bool, optional
Discard data in *ab* (may enhance performance)

overwrite_b : bool, optional
Discard data in *b* (may enhance performance)

lower : bool, optional
Is the matrix in the lower form. (Default is upper form)

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **x** : (M,) or (M, K) ndarray
The solution to the system a x = b. Shape of return matches shape of *b*.

`scipy.linalg.`**`solve_circulant`**(*c*, *b*, *singular='raise'*, *tol=None*, *caxis=-1*, *baxis=0*, *outaxis=0*)

Solve C x = b for x, where C is a circulant matrix.

C is the circulant matrix associated with the vector *c*.

The system is solved by doing division in Fourier space. The calculation is:

```
x = ifft(fft(b) / fft(c))
```

where *fft* and *ifft* are the fast Fourier transform and its inverse, respectively. For a large vector *c*, this is *much* faster than solving the system with the full circulant matrix.

Parameters **c** : array_like
The coefficients of the circulant matrix.

b : array_like
Right-hand side matrix in `a x = b`.

singular : str, optional
This argument controls how a near singular circulant matrix is handled. If *singular* is "raise" and the circulant matrix is near singular, a `LinAlgError` is raised. If *singular* is "lstsq", the least squares solution is returned. Default is "raise".

tol : float, optional
If any eigenvalue of the circulant matrix has an absolute value that is less than or equal to *tol*, the matrix is considered to be near singular. If not given, *tol* is set to:

```
tol = abs_eigs.max() * abs_eigs.size * np.finfo(np.float64).eps
```

where *abs_eigs* is the array of absolute values of the eigenvalues of the circulant matrix.

caxis : int
When *c* has dimension greater than 1, it is viewed as a collection of circulant vectors. In this case, *caxis* is the axis of *c* that holds the vectors of circulant coefficients.

baxis : int
When *b* has dimension greater than 1, it is viewed as a collection of vectors. In this case, *baxis* is the axis of *b* that holds the right-hand side vectors.

outaxis : int
When *c* or *b* are multidimensional, the value returned by `solve_circulant` is multidimensional. In this case, *outaxis* is the axis of the result that holds the solution vectors.

Returns **x** : ndarray
Solution to the system `C x = b`.

Raises **LinAlgError**
If the circulant matrix associated with *c* is near singular.

See also:

circulant

Notes

For a one-dimensional vector c with length m, and an array b with shape `(m, ...)`,

solve_circulant(c, b)

returns the same result as

solve(circulant(c), b)

where *solve* and *circulant* are from *scipy.linalg*.

New in version 0.16.0.

Examples

```
>>> from scipy.linalg import solve_circulant, solve, circulant, lstsq
```

```
>>> c = np.array([2, 2, 4])
>>> b = np.array([1, 2, 3])
>>> solve_circulant(c, b)
array([ 0.75, -0.25,  0.25])
```

Compare that result to solving the system with *scipy.linalg.solve*:

```
>>> solve(circulant(c), b)
array([ 0.75, -0.25,  0.25])
```

A singular example:

```
>>> c = np.array([1, 1, 0, 0])
>>> b = np.array([1, 2, 3, 4])
```

Calling `solve_circulant(c, b)` will raise a *LinAlgError*. For the least square solution, use the option `singular='lstsq'`:

```
>>> solve_circulant(c, b, singular='lstsq')
array([ 0.25,  1.25,  2.25,  1.25])
```

Compare to *scipy.linalg.lstsq*:

```
>>> x, resid, rnk, s = lstsq(circulant(c), b)
>>> x
array([ 0.25,  1.25,  2.25,  1.25])
```

A broadcasting example:

Suppose we have the vectors of two circulant matrices stored in an array with shape (2, 5), and three b vectors stored in an array with shape (3, 5). For example,

```
>>> c = np.array([[1.5, 2, 3, 0, 0], [1, 1, 4, 3, 2]])
>>> b = np.arange(15).reshape(-1, 5)
```

We want to solve all combinations of circulant matrices and b vectors, with the result stored in an array with shape (2, 3, 5). When we disregard the axes of c and b that hold the vectors of coefficients, the shapes of the collections are (2,) and (3,), respectively, which are not compatible for broadcasting. To have a broadcast result

with shape (2, 3), we add a trivial dimension to *c*: `c[:, np.newaxis, :]` has shape (2, 1, 5). The last dimension holds the coefficients of the circulant matrices, so when we call *solve_circulant*, we can use the default `caxis=-1`. The coefficients of the *b* vectors are in the last dimension of the array *b*, so we use `baxis=-1`. If we use the default *outaxis*, the result will have shape (5, 2, 3), so we'll use `outaxis=-1` to put the solution vectors in the last dimension.

```
>>> x = solve_circulant(c[:, np.newaxis, :], b, baxis=-1, outaxis=-1)
>>> x.shape
(2, 3, 5)
>>> np.set_printoptions(precision=3)  # For compact output of numbers.
>>> x
array([[[-0.118,  0.22 ,  1.277, -0.142,  0.302],
        [ 0.651,  0.989,  2.046,  0.627,  1.072],
        [ 1.42 ,  1.758,  2.816,  1.396,  1.841]],
       [[ 0.401,  0.304,  0.694, -0.867,  0.377],
        [ 0.856,  0.758,  1.149, -0.412,  0.831],
        [ 1.31 ,  1.213,  1.603,  0.042,  1.286]]])
```

Check by solving one pair of *c* and *b* vectors (cf. `x[1, 1, :]`):

```
>>> solve_circulant(c[1], b[1, :])
array([ 0.856,  0.758,  1.149, -0.412,  0.831])
```

`scipy.linalg.`**`solve_triangular`**(*a*, *b*, *trans=0*, *lower=False*, *unit_diagonal=False*, *overwrite_b=False*, *debug=None*, *check_finite=True*)

Solve the equation *a x = b* for *x*, assuming a is a triangular matrix.

Parameters

a : (M, M) array_like
: A triangular matrix

b : (M,) or (M, N) array_like
: Right-hand side matrix in *a x = b*

lower : bool, optional
: Use only data contained in the lower triangle of *a*. Default is to use upper triangle.

trans : {0, 1, 2, 'N', 'T', 'C'}, optional
: Type of system to solve:

trans	system
0 or 'N'	a x = b
1 or 'T'	a^T x = b
2 or 'C'	a^H x = b

unit_diagonal : bool, optional
: If True, diagonal elements of *a* are assumed to be 1 and will not be referenced.

overwrite_b : bool, optional
: Allow overwriting data in *b* (may enhance performance)

check_finite : bool, optional
: Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

x : (M,) or (M, N) ndarray
: Solution to the system *a x = b*. Shape of return matches *b*.

Raises

LinAlgError
: If *a* is singular

Notes

New in version 0.9.0.

`scipy.linalg.`**`solve_toeplitz`**`(`*c_or_cr*, *b*, *check_finite=True*`)`

Solve a Toeplitz system using Levinson Recursion

The Toeplitz matrix has constant diagonals, with c as its first column and r as its first row. If r is not given, `r == conjugate(c)` is assumed.

Parameters

c_or_cr : array_like or tuple of (array_like, array_like)
: The vector `c`, or a tuple of arrays (`c`, `r`). Whatever the actual shape of `c`, it will be converted to a 1-D array. If not supplied, `r = conjugate(c)` is assumed; in this case, if c[0] is real, the Toeplitz matrix is Hermitian. r[0] is ignored; the first row of the Toeplitz matrix is `[c[0], r[1:]]`. Whatever the actual shape of `r`, it will be converted to a 1-D array.

b : (M,) or (M, K) array_like
: Right-hand side in `T x = b`.

check_finite : bool, optional
: Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (result entirely NaNs) if the inputs do contain infinities or NaNs.

Returns

x : (M,) or (M, K) ndarray
: The solution to the system `T x = b`. Shape of return matches shape of *b*.

Notes

The solution is computed using Levinson-Durbin recursion, which is faster than generic least-squares methods, but can be less numerically stable.

`scipy.linalg.`**`det`**`(`*a*, *overwrite_a=False*, *check_finite=True*`)`

Compute the determinant of a matrix

The determinant of a square matrix is a value derived arithmetically from the coefficients of the matrix.

The determinant for a 3x3 matrix, for example, is computed as follows:

```
a    b    c
d    e    f = A
g    h    i

det(A) = a*e*i + b*f*g + c*d*h - c*e*g - b*d*i - a*f*h
```

Parameters

a : (M, M) array_like
: A square matrix.

overwrite_a : bool, optional
: Allow overwriting data in a (may enhance performance).

check_finite : bool, optional
: Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

det : float or complex
: Determinant of *a*.

Notes

The determinant is computed via LU factorization, LAPACK routine z/dgetrf.

Examples

```
>>> from scipy import linalg
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a)
0.0
>>> a = np.array([[0,2,3], [4,5,6], [7,8,9]])
>>> linalg.det(a)
3.0
```

`scipy.linalg.norm(a, ord=None, axis=None, keepdims=False)`

Matrix or vector norm.

This function is able to return one of seven different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

Parameters

a : (M,) or (M, N) array_like
Input array. If *axis* is None, *a* must be 1-D or 2-D.

ord : {non-zero int, inf, -inf, 'fro'}, optional
Order of the norm (see table under `Notes`). inf means numpy's *inf* object

axis : {int, 2-tuple of ints, None}, optional
If *axis* is an integer, it specifies the axis of *a* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *a* is 1-D) or a matrix norm (when *a* is 2-D) is returned.

keepdims : bool, optional
If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original *a*.

Returns

n : float or ndarray
Norm of the matrix or vector(s).

Notes

For values of `ord <= 0`, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes.

The following norms can be calculated:

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	–
inf	max(sum(abs(x), axis=1))	max(abs(x))
-inf	min(sum(abs(x), axis=1))	min(abs(x))
0	–	sum(x != 0)
1	max(sum(abs(x), axis=0))	as below
-1	min(sum(abs(x), axis=0))	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	–	sum(abs(x)**ord)**(1./ord)

The Frobenius norm is given by *[R111]*:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

The `axis` and `keepdims` arguments are passed directly to `numpy.linalg.norm` and are only usable if they are supported by the version of numpy in use.

References

[R111]

Examples

```
>>> from scipy.linalg import norm
>>> a = np.arange(9) - 4.0
>>> a
array([-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
>>> b = a.reshape((3, 3))
>>> b
array([[-4., -3., -2.],
       [-1.,  0.,  1.],
       [ 2.,  3.,  4.]])
```

```
>>> norm(a)
7.745966692414834
>>> norm(b)
7.745966692414834
>>> norm(b, 'fro')
7.745966692414834
>>> norm(a, np.inf)
4
>>> norm(b, np.inf)
9
>>> norm(a, -np.inf)
0
>>> norm(b, -np.inf)
2
```

```
>>> norm(a, 1)
20
>>> norm(b, 1)
7
>>> norm(a, -1)
-4.6566128774142013e-010
>>> norm(b, -1)
6
>>> norm(a, 2)
7.745966692414834
>>> norm(b, 2)
7.3484692283495345
```

```
>>> norm(a, -2)
0
>>> norm(b, -2)
1.8570331885190563e-016
>>> norm(a, 3)
5.8480354764257312
>>> norm(a, -3)
0
```

`scipy.linalg.`**`lstsq`**(*a*, *b*, *cond=None*, *overwrite_a=False*, *overwrite_b=False*, *check_finite=True*, *lapack_driver=None*)

Compute least-squares solution to equation Ax = b.

Compute a vector x such that the 2-norm `|b - A x|` is minimized.

Parameters
- **a** : (M, N) array_like
  Left hand side matrix (2-D array).
- **b** : (M,) or (M, K) array_like

Right hand side matrix or vector (1-D or 2-D array).

cond : float, optional
Cutoff for 'small' singular values; used to determine effective rank of a. Singular values smaller than `rcond * largest_singular_value` are considered zero.

overwrite_a : bool, optional
Discard data in *a* (may enhance performance). Default is False.

overwrite_b : bool, optional
Discard data in *b* (may enhance performance). Default is False.

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

lapack_driver: str, optional
Which LAPACK driver is used to solve the least-squares problem. Options are `'gelsd'`, `'gelsy'`, `'gelss'`. Default (`'gelsd'`) is a good choice. However, `'gelsy'` can be slightly faster on many problems. `'gelss'` was used historically. It is generally slow but uses less memory.
New in version 0.17.0.

Returns

x : (N,) or (N, K) ndarray
Least-squares solution. Return shape matches shape of *b*.

residues : () or (1,) or (K,) ndarray
Sums of residues, squared 2-norm for each column in `b - a x`. If rank of matrix a is `< N` or `> M`, or `'gelsy'` is used, this is an empty array. If b was 1-D, this is an (1,) shape array, otherwise the shape is (K,).

rank : int
Effective rank of matrix *a*.

s : (min(M,N),) ndarray or None
Singular values of *a*. The condition number of a is `abs(s[0] / s[-1])`. None is returned when `'gelsy'` is used.

Raises

LinAlgError
If computation does not converge.

ValueError
When parameters are wrong.

See also:

`optimize.nnls`
linear least squares with non-negativity constraint

`scipy.linalg.`**`pinv`**(*a*, *cond=None*, *rcond=None*, *return_rank=False*, *check_finite=True*)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate a generalized inverse of a matrix using a least-squares solver.

Parameters

a : (M, N) array_like
Matrix to be pseudo-inverted.

cond, rcond : float, optional
Cutoff for 'small' singular values in the least-squares solver. Singular values smaller than `rcond * largest_singular_value` are considered zero.

return_rank : bool, optional
if True, return the effective rank of the matrix

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns
B : (N, M) ndarray
The pseudo-inverse of matrix *a*.
rank : int
The effective rank of the matrix. Returned if return_rank == True

Raises
LinAlgError
If computation does not converge.

Examples

```
>>> from scipy import linalg
>>> a = np.random.randn(9, 6)
>>> B = linalg.pinv(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

`scipy.linalg.`**`pinv2`**(*a*, *cond=None*, *rcond=None*, *return_rank=False*, *check_finite=True*)
Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate a generalized inverse of a matrix using its singular-value decomposition and including all 'large' singular values.

Parameters
a : (M, N) array_like
Matrix to be pseudo-inverted.
cond, rcond : float or None
Cutoff for 'small' singular values. Singular values smaller than `rcond*largest_singular_value` are considered zero. If None or -1, suitable machine precision is used.
return_rank : bool, optional
if True, return the effective rank of the matrix
check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns
B : (N, M) ndarray
The pseudo-inverse of matrix *a*.
rank : int
The effective rank of the matrix. Returned if return_rank == True

Raises
LinAlgError
If SVD computation does not converge.

Examples

```
>>> from scipy import linalg
>>> a = np.random.randn(9, 6)
>>> B = linalg.pinv2(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

`scipy.linalg.`**`pinvh`**(*a*, *cond=None*, *rcond=None*, *lower=True*, *return_rank=False*, *check_finite=True*)
Compute the (Moore-Penrose) pseudo-inverse of a Hermitian matrix.

Calculate a generalized inverse of a Hermitian or real symmetric matrix using its eigenvalue decomposition and including all eigenvalues with 'large' absolute value.

Parameters

a : (N, N) array_like
: Real symmetric or complex hermetian matrix to be pseudo-inverted

cond, rcond : float or None
: Cutoff for 'small' eigenvalues. Singular values smaller than rcond * largest_eigenvalue are considered zero.
  If None or -1, suitable machine precision is used.

lower : bool, optional
: Whether the pertinent array data is taken from the lower or upper triangle of a. (Default: lower)

return_rank : bool, optional
: if True, return the effective rank of the matrix

check_finite : bool, optional
: Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

B : (N, N) ndarray
: The pseudo-inverse of matrix *a*.

rank : int
: The effective rank of the matrix. Returned if return_rank == True

Raises

LinAlgError
: If eigenvalue does not converge

Examples

```
>>> from scipy.linalg import pinvh
>>> a = np.random.randn(9, 6)
>>> a = np.dot(a, a.T)
>>> B = pinvh(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

`scipy.linalg.`**`kron`**(*a*, *b*)

Kronecker product.

The result is the block matrix:

```
a[0,0]*b    a[0,1]*b  ... a[0,-1]*b
a[1,0]*b    a[1,1]*b  ... a[1,-1]*b
...
a[-1,0]*b   a[-1,1]*b ... a[-1,-1]*b
```

Parameters

a : (M, N) ndarray
: Input array

b : (P, Q) ndarray
: Input array

Returns

A : (M*P, N*Q) ndarray
: Kronecker product of *a* and *b*.

Examples

```
>>> from numpy import array
>>> from scipy.linalg import kron
>>> kron(array([[1,2],[3,4]]), array([[1,1,1]]))
```

```
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
```

scipy.linalg.**tril**(*m*, *k=0*)

Make a copy of a matrix with elements above the k-th diagonal zeroed.

Parameters
m : array_like
Matrix whose elements to return
k : int, optional
Diagonal above which to zero elements. $k == 0$ is the main diagonal, $k < 0$ subdiagonal and $k > 0$ superdiagonal.

Returns
tril : ndarray
Return is the same shape and type as *m*.

Examples

```
>>> from scipy.linalg import tril
>>> tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 0,  0,  0],
       [ 4,  0,  0],
       [ 7,  8,  0],
       [10, 11, 12]])
```

scipy.linalg.**triu**(*m*, *k=0*)

Make a copy of a matrix with elements below the k-th diagonal zeroed.

Parameters
m : array_like
Matrix whose elements to return
k : int, optional
Diagonal below which to zero elements. $k == 0$ is the main diagonal, $k < 0$ subdiagonal and $k > 0$ superdiagonal.

Returns
triu : ndarray
Return matrix with zeroed elements below the k-th diagonal and has same shape and type as *m*.

Examples

```
>>> from scipy.linalg import triu
>>> triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 0,  8,  9],
       [ 0,  0, 12]])
```

scipy.linalg.**orthogonal_procrustes**(*A*, *B*, *check_finite=True*)

Compute the matrix solution of the orthogonal Procrustes problem.

Given matrices A and B of equal shape, find an orthogonal matrix R that most closely maps A to B *[R112]*. Note that unlike higher level Procrustes analyses of spatial data, this function only uses orthogonal transformations like rotations and reflections, and it does not use scaling or translation.

Parameters
A : (M, N) array_like
Matrix to be mapped.
B : (M, N) array_like
Target matrix.
check_finite : bool, optional

Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **R** : (N, N) ndarray
The matrix solution of the orthogonal Procrustes problem. Minimizes the Frobenius norm of dot(A, R) - B, subject to dot(R.T, R) == I.

scale : float
Sum of the singular values of `dot(A.T, B)`.

Raises **ValueError**
If the input arrays are incompatibly shaped. This may also be raised if matrix A or B contains an inf or nan and check_finite is True, or if the matrix product AB contains an inf or nan.

Notes

New in version 0.15.0.

References

[R112]

`scipy.linalg.`**`matrix_balance`**(*A*, *permute=True*, *scale=True*, *separate=False*, *overwrite_a=False*)

Compute a diagonal similarity transformation for row/column balancing.

The balancing tries to equalize the row and column 1-norms by applying a similarity transformation such that the magnitude variation of the matrix entries is reflected to the scaling matrices.

Moreover, if enabled, the matrix is first permuted to isolate the upper triangular parts of the matrix and, again if scaling is also enabled, only the remaining subblocks are subjected to scaling.

The balanced matrix satisfies the following equality

$$B = T^{-1}AT$$

The scaling coefficients are approximated to the nearest power of 2 to avoid round-off errors.

Parameters **A** : (n, n) array_like
Square data matrix for the balancing.

permute : bool, optional
The selector to define whether permutation of A is also performed prior to scaling.

scale : bool, optional
The selector to turn on and off the scaling. If False, the matrix will not be scaled.

separate : bool, optional
This switches from returning a full matrix of the transformation to a tuple of two separate 1D permutation and scaling arrays.

overwrite_a : bool, optional
This is passed to xGEBAL directly. Essentially, overwrites the result to the data. It might increase the space efficiency. See LAPACK manual for details. This is False by default.

Returns **B** : (n, n) ndarray
Balanced matrix

T : (n, n) ndarray
A possibly permuted diagonal matrix whose nonzero entries are integer powers of 2 to avoid numerical truncation errors.

scale, perm : (n,) ndarray
If `separate` keyword is set to True then instead of the array `T` above, the scaling and the permutation vectors are given separately as a tuple without allocating the full array `T`.

New in version 0.19.0.

Notes

This algorithm is particularly useful for eigenvalue and matrix decompositions and in many cases it is already called by various LAPACK routines.

The algorithm is based on the well-known technique of *[R108]* and has been modified to account for special cases. See *[R109]* for details which have been implemented since LAPACK v3.5.0. Before this version there are corner cases where balancing can actually worsen the conditioning. See *[R110]* for such examples.

The code is a wrapper around LAPACK's xGEBAL routine family for matrix balancing.

References

[R108], *[R109]*, *[R110]*

Examples

```
>>> from scipy import linalg
>>> x = np.array([[1,2,0], [9,1,0.01], [1,2,10*np.pi]])
```

```
>>> y, permscale = linalg.matrix_balance(x)
>>> np.abs(x).sum(axis=0) / np.abs(x).sum(axis=1)
array([ 3.66666667,  0.4995005 ,  0.91312162])
```

```
>>> np.abs(y).sum(axis=0) / np.abs(y).sum(axis=1) # 1-norms approx. equal
array([ 1.10625   ,  0.90547703,  1.00011878])
```

```
>>> permscale  # only powers of 2 (0.5 == 2^(-1))
array([[  0.5,   0. ,   0. ],
       [  0. ,   1. ,   0. ],
       [  0. ,   0. ,  16. ]])
```

exception `scipy.linalg.`**`LinAlgError`**

Generic Python-exception-derived object raised by linalg functions.

General purpose exception class, derived from Python's exception.Exception class, programmatically raised in linalg functions when a Linear Algebra-related condition would prevent further correct execution of the function.

Parameters **None**

Examples

```
>>> from numpy import linalg as LA
>>> LA.inv(np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...linalg.py", line 350,
    in inv return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
  File "...linalg.py", line 249,
    in solve
    raise LinAlgError('Singular matrix')
numpy.linalg.LinAlgError: Singular matrix
```

5.9.2 Eigenvalue Problems

eig(a[, b, left, right, overwrite_a, ...])	Solve an ordinary or generalized eigenvalue problem of a square matrix.
eigvals(a[, b, overwrite_a, check_finite, ...])	Compute eigenvalues from an ordinary or generalized eigenvalue problem.
eigh(a[, b, lower, eigvals_only, ...])	Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
eigvalsh(a[, b, lower, overwrite_a, ...])	Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.
eig_banded(a_band[, lower, eigvals_only, ...])	Solve real symmetric or complex hermitian band matrix eigenvalue problem.
eigvals_banded(a_band[, lower, ...])	Solve real symmetric or complex hermitian band matrix eigenvalue problem.

scipy.linalg.**eig**(*a*, *b=None*, *left=False*, *right=True*, *overwrite_a=False*, *overwrite_b=False*, *check_finite=True*, *homogeneous_eigvals=False*)

Solve an ordinary or generalized eigenvalue problem of a square matrix.

Find eigenvalues w and right or left eigenvectors of a general matrix:

```
a   vr[:,i] = w[i]        b   vr[:,i]
a.H vl[:,i] = w[i].conj() b.H vl[:,i]
```

where `.H` is the Hermitian conjugation.

Parameters

a : (M, M) array_like
A complex or real matrix whose eigenvalues and eigenvectors will be computed.

b : (M, M) array_like, optional
Right-hand side matrix in a generalized eigenvalue problem. Default is None, identity matrix is assumed.

left : bool, optional
Whether to calculate and return left eigenvectors. Default is False.

right : bool, optional
Whether to calculate and return right eigenvectors. Default is True.

overwrite_a : bool, optional
Whether to overwrite *a*; may improve performance. Default is False.

overwrite_b : bool, optional
Whether to overwrite *b*; may improve performance. Default is False.

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

homogeneous_eigvals : bool, optional
If True, return the eigenvalues in homogeneous coordinates. In this case `w` is a (2, M) array so that:

```
w[1,i] a vr[:,i] = w[0,i] b vr[:,i]
```

Default is False.

Returns

w : (M,) or (2, M) double or complex ndarray
The eigenvalues, each repeated according to its multiplicity. The shape is (M,) unless `homogeneous_eigvals=True`.

vl : (M, M) double or complex ndarray
The normalized left eigenvector corresponding to the eigenvalue `w[i]` is the column vl[:,i]. Only returned if `left=True`.

vr : (M, M) double or complex ndarray
The normalized right eigenvector corresponding to the eigenvalue `w[i]` is the column `vr[:,i]`. Only returned if `right=True`.

Raises **LinAlgError**
If eigenvalue computation does not converge.

See also:

eigh Eigenvalues and right eigenvectors for symmetric/Hermitian arrays.

`scipy.linalg.`**`eigvals`**(*a*, *b=None*, *overwrite_a=False*, *check_finite=True*, *homogeneous_eigvals=False*)

Compute eigenvalues from an ordinary or generalized eigenvalue problem.

Find eigenvalues of a general matrix:

```
a   vr[:,i] = w[i]        b   vr[:,i]
```

Parameters

a : (M, M) array_like
A complex or real matrix whose eigenvalues and eigenvectors will be computed.

b : (M, M) array_like, optional
Right-hand side matrix in a generalized eigenvalue problem. If omitted, identity matrix is assumed.

overwrite_a : bool, optional
Whether to overwrite data in a (may improve performance)

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

homogeneous_eigvals : bool, optional
If True, return the eigenvalues in homogeneous coordinates. In this case `w` is a (2, M) array so that:

```
w[1,i] a vr[:,i] = w[0,i] b vr[:,i]
```

Default is False.

Returns **w** : (M,) or (2, M) double or complex ndarray
The eigenvalues, each repeated according to its multiplicity but not in any specific order. The shape is (M,) unless `homogeneous_eigvals=True`.

Raises **LinAlgError**
If eigenvalue computation does not converge

See also:

eigvalsh eigenvalues of symmetric or Hermitian arrays,

eig eigenvalues and right eigenvectors of general arrays.

eigh eigenvalues and eigenvectors of symmetric/Hermitian arrays.

`scipy.linalg.`**`eigh`**(*a*, *b=None*, *lower=True*, *eigvals_only=False*, *overwrite_a=False*, *overwrite_b=False*, *turbo=True*, *eigvals=None*, *type=1*, *check_finite=True*)

Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues w and optionally eigenvectors v of matrix *a*, where *b* is positive definite:

```
              a v[:,i] = w[i] b v[:,i]
v[i,:].conj() a v[:,i] = w[i]
v[i,:].conj() b v[:,i] = 1
```

Parameters **a** : (M, M) array_like

A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.

b : (M, M) array_like, optional

A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.

lower : bool, optional

Whether the pertinent array data is taken from the lower or upper triangle of *a*. (Default: lower)

eigvals_only : bool, optional

Whether to calculate only eigenvalues and no eigenvectors. (Default: both are calculated)

turbo : bool, optional

Use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if eigvals=None)

eigvals : tuple (lo, hi), optional

Indexes of the smallest and largest (in ascending order) eigenvalues and corresponding eigenvectors to be returned: 0 <= lo <= hi <= M-1. If omitted, all eigenvalues and eigenvectors are returned.

type : int, optional

Specifies the problem type to be solved:

type = 1: a v[:,i] = w[i] b v[:,i]
type = 2: a b v[:,i] = w[i] v[:,i]
type = 3: b a v[:,i] = w[i] v[:,i]

overwrite_a : bool, optional

Whether to overwrite data in *a* (may improve performance)

overwrite_b : bool, optional

Whether to overwrite data in *b* (may improve performance)

check_finite : bool, optional

Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **w** : (N,) float ndarray

The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.

v : (M, N) complex ndarray

(if eigvals_only == False)

The normalized selected eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].

Normalization:

type 1 and 3: v.conj() a v = w
type 2: inv(v).conj() a inv(v) = w
type = 1 or 2: v.conj() b v = I
type = 3: v.conj() inv(b) v = I

Raises **LinAlgError**

If eigenvalue computation does not converge, an error occurred, or b matrix is not definite positive. Note that if input matrices are not symmetric or hermitian, no error is reported but results will be wrong.

See also:

eig eigenvalues and right eigenvectors for non-symmetric arrays

scipy.linalg.**eigvalsh**(*a*, *b=None*, *lower=True*, *overwrite_a=False*, *overwrite_b=False*, *turbo=True*, *eigvals=None*, *type=1*, *check_finite=True*)

Solve an ordinary or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues w of matrix a, where b is positive definite:

```
                a v[:,i] = w[i] b v[:,i]
v[i,:].conj() a v[:,i] = w[i]
v[i,:].conj() b v[:,i] = 1
```

Parameters

a : (M, M) array_like
: A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.

b : (M, M) array_like, optional
: A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.

lower : bool, optional
: Whether the pertinent array data is taken from the lower or upper triangle of *a*. (Default: lower)

turbo : bool, optional
: Use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if eigvals=None)

eigvals : tuple (lo, hi), optional
: Indexes of the smallest and largest (in ascending order) eigenvalues and corresponding eigenvectors to be returned: 0 <= lo < hi <= M-1. If omitted, all eigenvalues and eigenvectors are returned.

type : int, optional
: Specifies the problem type to be solved:
  type = 1: a v[:,i] = w[i] b v[:,i]
  type = 2: a b v[:,i] = w[i] v[:,i]
  type = 3: b a v[:,i] = w[i] v[:,i]

overwrite_a : bool, optional
: Whether to overwrite data in *a* (may improve performance)

overwrite_b : bool, optional
: Whether to overwrite data in *b* (may improve performance)

check_finite : bool, optional
: Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

w : (N,) float ndarray
: The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.

Raises

LinAlgError
: If eigenvalue computation does not converge, an error occurred, or b matrix is not definite positive. Note that if input matrices are not symmetric or hermitian, no error is reported but results will be wrong.

See also:

eigvals eigenvalues of general arrays

eigh eigenvalues and right eigenvectors for symmetric/Hermitian arrays

eig eigenvalues and right eigenvectors for non-symmetric arrays

scipy.linalg.**eig_banded**(*a_band*, *lower=False*, *eigvals_only=False*, *overwrite_a_band=False*, *select='a'*, *select_range=None*, *max_ev=0*, *check_finite=True*)

Solve real symmetric or complex hermitian band matrix eigenvalue problem.

Find eigenvalues w and optionally right eigenvectors v of a:

```
a v[:,i] = w[i] v[:,i]
v.H v    = identity
```

The matrix a is stored in a_band either in lower diagonal or upper diagonal ordered form:

a_band[u + i - j, j] == a[i,j] (if upper form; i <= j) a_band[i - j, j] == a[i,j] (if lower form; i >= j)

where u is the number of bands above the diagonal.

Example of a_band (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters

a_band : (u+1, M) array_like
: The bands of the M by M matrix a.

lower : bool, optional
: Is the matrix in the lower form. (Default is upper form)

eigvals_only : bool, optional
: Compute only the eigenvalues and no eigenvectors. (Default: calculate also eigenvectors)

overwrite_a_band : bool, optional
: Discard data in a_band (may enhance performance)

select : {'a', 'v', 'i'}, optional
: Which eigenvalues to calculate

select	calculated
'a'	All eigenvalues
'v'	Eigenvalues in the interval (min, max]
'i'	Eigenvalues with indices min <= i <= max

select_range : (min, max), optional
: Range of selected eigenvalues

max_ev : int, optional
: For select=='v', maximum number of eigenvalues expected. For other values of select, has no meaning.
: In doubt, leave this parameter untouched.

check_finite : bool, optional
: Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **w** : (M,) ndarray
The eigenvalues, in ascending order, each repeated according to its multiplicity.
v : (M, M) float or complex ndarray
The normalized eigenvector corresponding to the eigenvalue w[i] is the column v[:,i].
Raises LinAlgError if eigenvalue computation does not converge

`scipy.linalg.`**`eigvals_banded`**(*a_band*, *lower=False*, *overwrite_a_band=False*, *select='a'*, *select_range=None*, *check_finite=True*)

Solve real symmetric or complex hermitian band matrix eigenvalue problem.

Find eigenvalues w of a:

```
a v[:,i] = w[i] v[:,i]
v.H v    = identity
```

The matrix a is stored in a_band either in lower diagonal or upper diagonal ordered form:

a_band[u + i - j, j] == a[i,j] (if upper form; i <= j) a_band[i - j, j] == a[i,j] (if lower form; i >= j)

where u is the number of bands above the diagonal.

Example of a_band (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Cells marked with * are not used.

Parameters **a_band** : (u+1, M) array_like
The bands of the M by M matrix a.
lower : bool, optional
Is the matrix in the lower form. (Default is upper form)
overwrite_a_band : bool, optional
Discard data in a_band (may enhance performance)
select : {'a', 'v', 'i'}, optional
Which eigenvalues to calculate

select	calculated
'a'	All eigenvalues
'v'	Eigenvalues in the interval (min, max]
'i'	Eigenvalues with indices min <= i <= max

select_range : (min, max), optional
Range of selected eigenvalues
check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **w** : (M,) ndarray
The eigenvalues, in ascending order, each repeated according to its multiplicity.
Raises LinAlgError if eigenvalue computation does not converge

See also:

eig_banded eigenvalues and right eigenvectors for symmetric/Hermitian band matrices

eigvals eigenvalues of general arrays

eigh eigenvalues and right eigenvectors for symmetric/Hermitian arrays

eig eigenvalues and right eigenvectors for non-symmetric arrays

5.9.3 Decompositions

lu(a[, permute_l, overwrite_a, check_finite])	Compute pivoted LU decomposition of a matrix.
lu_factor(a[, overwrite_a, check_finite])	Compute pivoted LU decomposition of a matrix.
lu_solve(lu_and_piv, b[, trans, ...])	Solve an equation system, a x = b, given the LU factorization of a
svd(a[, full_matrices, compute_uv, ...])	Singular Value Decomposition.
svdvals(a[, overwrite_a, check_finite])	Compute singular values of a matrix.
diagsvd(s, M, N)	Construct the sigma matrix in SVD from singular values and size M, N.
orth(A)	Construct an orthonormal basis for the range of A using SVD
cholesky(a[, lower, overwrite_a, check_finite])	Compute the Cholesky decomposition of a matrix.
cholesky_banded(ab[, overwrite_ab, lower, ...])	Cholesky decompose a banded Hermitian positive-definite matrix
cho_factor(a[, lower, overwrite_a, check_finite])	Compute the Cholesky decomposition of a matrix, to use in cho_solve
cho_solve(c_and_lower, b[, overwrite_b, ...])	Solve the linear equations A x = b, given the Cholesky factorization of A.
cho_solve_banded(cb_and_lower, b[, ...])	Solve the linear equations A x = b, given the Cholesky factorization of A.
polar(a[, side])	Compute the polar decomposition.
qr(a[, overwrite_a, lwork, mode, pivoting, ...])	Compute QR decomposition of a matrix.
qr_multiply(a, c[, mode, pivoting, ...])	Calculate the QR decomposition and multiply Q with a matrix.
qr_update(Q, R, u, v[, overwrite_qruv, ...])	Rank-k QR update
qr_delete(Q, R, k, int p=1[, which, ...])	QR downdate on row or column deletions
qr_insert(Q, R, u, k[, which, rcond, ...])	QR update on row or column insertions
rq(a[, overwrite_a, lwork, mode, check_finite])	Compute RQ decomposition of a matrix.
qz(A, B[, output, lwork, sort, overwrite_a, ...])	QZ decomposition for generalized eigenvalues of a pair of matrices.
ordqz(A, B[, sort, output, overwrite_a, ...])	QZ decomposition for a pair of matrices with reordering.
schur(a[, output, lwork, overwrite_a, sort, ...])	Compute Schur decomposition of a matrix.
rsf2csf(T, Z[, check_finite])	Convert real Schur form to complex Schur form.
hessenberg(a[, calc_q, overwrite_a, ...])	Compute Hessenberg form of a matrix.

scipy.linalg.**lu**(*a*, *permute_l=False*, *overwrite_a=False*, *check_finite=True*)

Compute pivoted LU decomposition of a matrix.

The decomposition is:

```
A = P L U
```

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Parameters **a** : (M, N) array_like

Array to decompose

permute_l : bool, optional
Perform the multiplication P*L (Default: do not permute)

overwrite_a : bool, optional
Whether to overwrite data in a (may improve performance)

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **(If permute_l == False)**

p : (M, M) ndarray
Permutation matrix

l : (M, K) ndarray
Lower triangular or trapezoidal matrix with unit diagonal. K = min(M, N)

u : (K, N) ndarray
Upper triangular or trapezoidal matrix

(If permute_l == True)

pl : (M, K) ndarray
Permuted L matrix. K = min(M, N)

u : (K, N) ndarray
Upper triangular or trapezoidal matrix

Notes

This is a LU factorization routine written for Scipy.

scipy.linalg.**lu_factor**(*a*, *overwrite_a=False*, *check_finite=True*)

Compute pivoted LU decomposition of a matrix.

The decomposition is:

```
A = P L U
```

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Parameters **a** : (M, M) array_like
Matrix to decompose

overwrite_a : bool, optional
Whether to overwrite data in A (may increase performance)

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **lu** : (N, N) ndarray
Matrix containing U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored.

piv : (N,) ndarray
Pivot indices representing the permutation matrix P: row i of matrix was interchanged with row piv[i].

See also:

lu_solve solve an equation system using the LU factorization of a matrix

Notes

This is a wrapper to the `*GETRF` routines from LAPACK.

`scipy.linalg.`**`lu_solve`**(*lu_and_piv*, *b*, *trans=0*, *overwrite_b=False*, *check_finite=True*)

Solve an equation system, a x = b, given the LU factorization of a

Parameters

(lu, piv)
: Factorization of the coefficient matrix a, as given by lu_factor

b : array
: Right-hand side

trans : {0, 1, 2}, optional
: Type of system to solve:

trans	system
0	a x = b
1	a^T x = b
2	a^H x = b

overwrite_b : bool, optional
: Whether to overwrite data in b (may increase performance)

check_finite : bool, optional
: Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

x : array
: Solution to the system

See also:

`lu_factor` LU factorize a matrix

`scipy.linalg.`**`svd`**(*a*, *full_matrices=True*, *compute_uv=True*, *overwrite_a=False*, *check_finite=True*, *lapack_driver='gesdd'*)

Singular Value Decomposition.

Factorizes the matrix a into two unitary matrices U and Vh, and a 1-D array s of singular values (real, non-negative) such that `a == U*S*Vh`, where S is a suitably shaped matrix of zeros with main diagonal s.

Parameters

a : (M, N) array_like
: Matrix to decompose.

full_matrices : bool, optional
: If True, *U* and *Vh* are of shape `(M,M)`, `(N,N)`. If False, the shapes are `(M,K)` and `(K,N)`, where `K = min(M,N)`.

compute_uv : bool, optional
: Whether to compute also *U* and *Vh* in addition to `s`. Default is True.

overwrite_a : bool, optional
: Whether to overwrite *a*; may improve performance. Default is False.

check_finite : bool, optional
: Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

lapack_driver : {'gesdd', 'gesvd'}, optional
: Whether to use the more efficient divide-and-conquer approach (`'gesdd'`) or general rectangular approach (`'gesvd'`) to compute the SVD. MATLAB and Octave use the `'gesvd'` approach. Default is `'gesdd'`.
: New in version 0.18.

Returns

U : ndarray

Unitary matrix having left singular vectors as columns. Of shape `(M,M)` or `(M,K)`, depending on *full_matrices*.

s : ndarray
The singular values, sorted in non-increasing order. Of shape (K,), with `K = min(M, N)`.

Vh : ndarray
Unitary matrix having right singular vectors as rows. Of shape `(N,N)` or `(K,N)` depending on *full_matrices*.

For `compute_uv=False`, only `s` is returned.

Raises **LinAlgError**
If SVD computation does not converge.

See also:

svdvals Compute singular values of a matrix.

diagsvd Construct the Sigma matrix, given the vector s.

Examples

```
>>> from scipy import linalg
>>> a = np.random.randn(9, 6) + 1.j*np.random.randn(9, 6)
>>> U, s, Vh = linalg.svd(a)
>>> U.shape, Vh.shape, s.shape
((9, 9), (6, 6), (6,))
```

```
>>> U, s, Vh = linalg.svd(a, full_matrices=False)
>>> U.shape, Vh.shape, s.shape
((9, 6), (6, 6), (6,))
>>> S = linalg.diagsvd(s, 6, 6)
>>> np.allclose(a, np.dot(U, np.dot(S, Vh)))
True
```

```
>>> s2 = linalg.svd(a, compute_uv=False)
>>> np.allclose(s, s2)
True
```

`scipy.linalg.`**`svdvals`**(*a*, *overwrite_a=False*, *check_finite=True*)
Compute singular values of a matrix.

Parameters **a** : (M, N) array_like
Matrix to decompose.

overwrite_a : bool, optional
Whether to overwrite *a*; may improve performance. Default is False.

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **s** : (min(M, N),) ndarray
The singular values, sorted in decreasing order.

Raises **LinAlgError**
If SVD computation does not converge.

See also:

svd Compute the full singular value decomposition of a matrix.

diagsvd Construct the Sigma matrix, given the vector s.

Notes

`svdvals(a)` only differs from `svd(a, compute_uv=False)` by its handling of the edge case of empty `a`, where it returns an empty sequence:

```
>>> a = np.empty((0, 2))
>>> from scipy.linalg import svdvals
>>> svdvals(a)
array([], dtype=float64)
```

`scipy.linalg.`**diagsvd**(*s*, *M*, *N*)

Construct the sigma matrix in SVD from singular values and size M, N.

Parameters **s** : (M,) or (N,) array_like

Singular values

M : int

Size of the matrix whose singular values are `s`.

N : int

Size of the matrix whose singular values are `s`.

Returns **S** : (M, N) ndarray

The S-matrix in the singular value decomposition

`scipy.linalg.`**orth**(*A*)

Construct an orthonormal basis for the range of A using SVD

Parameters **A** : (M, N) array_like

Input array

Returns **Q** : (M, K) ndarray

Orthonormal basis for the range of A. K = effective rank of A, as determined by automatic cutoff

See also:

svd Singular value decomposition of a matrix

`scipy.linalg.`**cholesky**(*a*, *lower=False*, *overwrite_a=False*, *check_finite=True*)

Compute the Cholesky decomposition of a matrix.

Returns the Cholesky decomposition, $A = LL^*$ or $A = U^*U$ of a Hermitian positive-definite matrix A.

Parameters **a** : (M, M) array_like

Matrix to be decomposed

lower : bool, optional

Whether to compute the upper or lower triangular Cholesky factorization. Default is upper-triangular.

overwrite_a : bool, optional

Whether to overwrite data in *a* (may improve performance).

check_finite : bool, optional

Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **c** : (M, M) ndarray

Upper- or lower-triangular Cholesky factor of *a*.

Raises **LinAlgError** : if decomposition fails.

Examples

```
>>> from scipy import array, linalg, dot
>>> a = array([[1,-2j],[2j,5]])
>>> L = linalg.cholesky(a, lower=True)
>>> L
array([[ 1.+0.j,  0.+0.j],
       [ 0.+2.j,  1.+0.j]])
>>> dot(L, L.T.conj())
array([[ 1.+0.j,  0.-2.j],
       [ 0.+2.j,  5.+0.j]])
```

`scipy.linalg.`**`cholesky_banded`**(*ab*, *overwrite_ab=False*, *lower=False*, *check_finite=True*)

Cholesky decompose a banded Hermitian positive-definite matrix

The matrix a is stored in ab either in lower diagonal or upper diagonal ordered form:

```
ab[u + i - j, j] == a[i,j]        (if upper form; i <= j)
ab[    i - j, j] == a[i,j]        (if lower form; i >= j)
```

Example of ab (shape of a is (6,6), u=2):

```
upper form:
*   *   a02 a13 a24 a35
*   a01 a12 a23 a34 a45
a00 a11 a22 a33 a44 a55

lower form:
a00 a11 a22 a33 a44 a55
a10 a21 a32 a43 a54 *
a20 a31 a42 a53 *   *
```

Parameters

ab : (u + 1, M) array_like
    Banded matrix

overwrite_ab : bool, optional
    Discard data in ab (may enhance performance)

lower : bool, optional
    Is the matrix in the lower form. (Default is upper form)

check_finite : bool, optional
    Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

c : (u + 1, M) ndarray
    Cholesky factorization of a, in the same banded format as ab

`scipy.linalg.`**`cho_factor`**(*a*, *lower=False*, *overwrite_a=False*, *check_finite=True*)

Compute the Cholesky decomposition of a matrix, to use in cho_solve

Returns a matrix containing the Cholesky decomposition, `A = L L*` or `A = U* U` of a Hermitian positive-definite matrix *a*. The return value can be directly used as the first parameter to cho_solve.

> **Warning:** The returned matrix also contains random data in the entries not used by the Cholesky decomposition. If you need to zero these entries, use the function *`cholesky`* instead.

Parameters

a : (M, M) array_like

Matrix to be decomposed

lower : bool, optional
Whether to compute the upper or lower triangular Cholesky factorization (Default: upper-triangular)

overwrite_a : bool, optional
Whether to overwrite data in a (may improve performance)

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **c** : (M, M) ndarray
Matrix whose upper or lower triangle contains the Cholesky factor of *a*. Other parts of the matrix contain random data.

lower : bool
Flag indicating whether the factor is in the lower or upper triangle

Raises **LinAlgError**
Raised if decomposition fails.

See also:

`cho_solve` Solve a linear set equations using the Cholesky factorization of a matrix.

`scipy.linalg.`**`cho_solve`**(*c_and_lower*, *b*, *overwrite_b=False*, *check_finite=True*)

Solve the linear equations A x = b, given the Cholesky factorization of A.

Parameters **(c, lower)** : tuple, (array, bool)
Cholesky factorization of a, as given by cho_factor

b : array
Right-hand side

overwrite_b : bool, optional
Whether to overwrite data in b (may improve performance)

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **x** : array
The solution to the system A x = b

See also:

`cho_factor`Cholesky factorization of a matrix

`scipy.linalg.`**`cho_solve_banded`**(*cb_and_lower*, *b*, *overwrite_b=False*, *check_finite=True*)

Solve the linear equations A x = b, given the Cholesky factorization of A.

Parameters **(cb, lower)** : tuple, (array, bool)
cb is the Cholesky factorization of A, as given by cholesky_banded. *lower* must be the same value that was given to cholesky_banded.

b : array
Right-hand side

overwrite_b : bool, optional
If True, the function will overwrite the values in *b*.

check_finite : bool, optional

Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **x** : array

The solution to the system A x = b

See also:

cholesky_banded
Cholesky factorization of a banded matrix

Notes

New in version 0.8.0.

`scipy.linalg.polar(a, side='right')`

Compute the polar decomposition.

Returns the factors of the polar decomposition *[R113]* *u* and *p* such that `a = up` (if *side* is "right") or `a = pu` (if *side* is "left"), where *p* is positive semidefinite. Depending on the shape of *a*, either the rows or columns of *u* are orthonormal. When *a* is a square array, *u* is a square unitary array. When *a* is not square, the "canonical polar decomposition" *[R114]* is computed.

Parameters **a** : (m, n) array_like

The array to be factored.

side : {'left', 'right'}, optional

Determines whether a right or left polar decomposition is computed. If *side* is "right", then `a = up`. If *side* is "left", then `a = pu`. The default is "right".

Returns **u** : (m, n) ndarray

If *a* is square, then *u* is unitary. If m > n, then the columns of *a* are orthonormal, and if m < n, then the rows of *u* are orthonormal.

p : ndarray

p is Hermitian positive semidefinite. If *a* is nonsingular, *p* is positive definite. The shape of *p* is (n, n) or (m, m), depending on whether *side* is "right" or "left", respectively.

References

[R113], *[R114]*

Examples

```
>>> from scipy.linalg import polar
>>> a = np.array([[1, -1], [2, 4]])
>>> u, p = polar(a)
>>> u
array([[ 0.85749293, -0.51449576],
       [ 0.51449576,  0.85749293]])
>>> p
array([[ 1.88648444,  1.2004901 ],
       [ 1.2004901 ,  3.94446746]])
```

A non-square example, with m < n:

```
>>> b = np.array([[0.5, 1, 2], [1.5, 3, 4]])
>>> u, p = polar(b)
>>> u
array([[-0.21196618, -0.42393237,  0.88054056],
       [ 0.39378971,  0.78757942,  0.4739708 ]])
>>> p
```

```
array([[ 0.48470147,  0.96940295,  1.15122648],
       [ 0.96940295,  1.9388059 ,  2.30245295],
       [ 1.15122648,  2.30245295,  3.65696431]])
>>> u.dot(p)   # Verify the decomposition.
array([[ 0.5,  1. ,  2. ],
       [ 1.5,  3. ,  4. ]])
>>> u.dot(u.T)   # The rows of u are orthonormal.
array([[  1.00000000e+00,  -2.07353665e-17],
       [ -2.07353665e-17,   1.00000000e+00]])
```

Another non-square example, with m > n:

```
>>> c = b.T
>>> u, p = polar(c)
>>> u
array([[-0.21196618,  0.39378971],
       [-0.42393237,  0.78757942],
       [ 0.88054056,  0.4739708 ]])
>>> p
array([[ 1.23116567,  1.93241587],
       [ 1.93241587,  4.84930602]])
>>> u.dot(p)   # Verify the decomposition.
array([[ 0.5,  1.5],
       [ 1. ,  3. ],
       [ 2. ,  4. ]])
>>> u.T.dot(u)  # The columns of u are orthonormal.
array([[  1.00000000e+00,  -1.26363763e-16],
       [ -1.26363763e-16,   1.00000000e+00]])
```

`scipy.linalg.`**`qr`**(*a*, *overwrite_a=False*, *lwork=None*, *mode='full'*, *pivoting=False*, *check_finite=True*)

Compute QR decomposition of a matrix.

Calculate the decomposition `A = Q R` where Q is unitary/orthogonal and R upper triangular.

Parameters

a : (M, N) array_like
  Matrix to be decomposed

overwrite_a : bool, optional
  Whether data in a is overwritten (may improve performance)

lwork : int, optional
  Work array size, lwork >= a.shape[1]. If None or -1, an optimal size is computed.

mode : {'full', 'r', 'economic', 'raw'}, optional
  Determines what information is to be returned: either both Q and R ('full', default), only R ('r') or both Q and R but computed in economy-size ('economic', see Notes). The final option 'raw' (added in Scipy 0.11) makes the function return two matrices (Q, TAU) in the internal format used by LAPACK.

pivoting : bool, optional
  Whether or not factorization should include pivoting for rank-revealing qr decomposition. If pivoting, compute the decomposition `A P = Q R` as above, but where P is chosen such that the diagonal of R is non-increasing.

check_finite : bool, optional
  Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

Q : float or complex ndarray
  Of shape (M, M), or (M, K) for `mode='economic'`. Not returned if `mode='r'`.

R : float or complex ndarray
  Of shape (M, N), or (K, N) for `mode='economic'`. `K = min(M, N)`.

P : int ndarray
Of shape (N,) for `pivoting=True`. Not returned if `pivoting=False`.

Raises **LinAlgError**
Raised if decomposition fails

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dorgqr, zungqr, dgeqp3, and zgeqp3.

If `mode=economic`, the shapes of Q and R are (M, K) and (K, N) instead of (M,M) and (M,N), with `K=min(M,N)`.

Examples

```
>>> from scipy import random, linalg, dot, diag, all, allclose
>>> a = random.randn(9, 6)
```

```
>>> q, r = linalg.qr(a)
>>> allclose(a, np.dot(q, r))
True
>>> q.shape, r.shape
((9, 9), (9, 6))
```

```
>>> r2 = linalg.qr(a, mode='r')
>>> allclose(r, r2)
True
```

```
>>> q3, r3 = linalg.qr(a, mode='economic')
>>> q3.shape, r3.shape
((9, 6), (6, 6))
```

```
>>> q4, r4, p4 = linalg.qr(a, pivoting=True)
>>> d = abs(diag(r4))
>>> all(d[1:] <= d[:-1])
True
>>> allclose(a[:, p4], dot(q4, r4))
True
>>> q4.shape, r4.shape, p4.shape
((9, 9), (9, 6), (6,))
```

```
>>> q5, r5, p5 = linalg.qr(a, mode='economic', pivoting=True)
>>> q5.shape, r5.shape, p5.shape
((9, 6), (6, 6), (6,))
```

`scipy.linalg.`**`qr_multiply`**(*a*, *c*, *mode='right'*, *pivoting=False*, *conjugate=False*, *overwrite_a=False*, *overwrite_c=False*)

Calculate the QR decomposition and multiply Q with a matrix.

Calculate the decomposition `A = Q R` where Q is unitary/orthogonal and R upper triangular. Multiply Q with a vector or a matrix c.

Parameters **a** : array_like, shape (M, N)
Matrix to be decomposed

c : array_like, one- or two-dimensional
calculate the product of c and q, depending on the mode:

mode : {'left', 'right'}, optional

`dot(Q, c)` is returned if mode is 'left', `dot(c, Q)` is returned if mode is 'right'. The shape of c must be appropriate for the matrix multiplications, if mode is 'left', `min(a.shape) == c.shape[0]`, if mode is 'right', `a.shape[0] == c.shape[1]`.

pivoting : bool, optional
: Whether or not factorization should include pivoting for rank-revealing qr decomposition, see the documentation of qr.

conjugate : bool, optional
: Whether Q should be complex-conjugated. This might be faster than explicit conjugation.

overwrite_a : bool, optional
: Whether data in a is overwritten (may improve performance)

overwrite_c : bool, optional
: Whether data in c is overwritten (may improve performance). If this is used, c must be big enough to keep the result, i.e. c.shape[0] = a.shape[0] if mode is 'left'.

Returns **CQ** : float or complex ndarray
: the product of Q and c, as defined in mode

R : float or complex ndarray
: Of shape (K, N), `K = min(M, N)`.

P : ndarray of ints
: Of shape (N,) for `pivoting=True`. Not returned if `pivoting=False`.

Raises **LinAlgError**
: Raised if decomposition fails

Notes

This is an interface to the LAPACK routines dgeqrf, zgeqrf, dormqr, zunmqr, dgeqp3, and zgeqp3.

New in version 0.11.0.

`scipy.linalg.`**`qr_update`**(*Q*, *R*, *u*, *v*, *overwrite_qruv=False*, *check_finite=True*)

Rank-k QR update

If `A = Q R` is the QR factorization of `A`, return the QR factorization of `A + u v**T` for real `A` or `A + u v**H` for complex `A`.

Parameters **Q** : (M, M) or (M, N) array_like
: Unitary/orthogonal matrix from the qr decomposition of A.

R : (M, N) or (N, N) array_like
: Upper triangular matrix from the qr decomposition of A.

u : (M,) or (M, k) array_like
: Left update vector

v : (N,) or (N, k) array_like
: Right update vector

overwrite_qruv : bool, optional
: If True, consume Q, R, u, and v, if possible, while performing the update, otherwise make copies as necessary. Defaults to False.

check_finite : bool, optional
: Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is True.

Returns **Q1** : ndarray
: Updated unitary/orthogonal factor

R1 : ndarray
: Updated upper triangular factor

See also:

qr, *qr_multiply*, *qr_delete*, *qr_insert*

Notes

This routine does not guarantee that the diagonal entries of *R1* are real or positive.

New in version 0.16.0.

References

[R121], [R122], [R123]

Examples

```
>>> from scipy import linalg
>>> a = np.array([[  3.,  -2.,  -2.],
...               [  6.,  -9.,  -3.],
...               [ -3.,  10.,   1.],
...               [  6.,  -7.,   4.],
...               [  7.,   8.,  -6.]])
>>> q, r = linalg.qr(a)
```

Given this q, r decomposition, perform a rank 1 update.

```
>>> u = np.array([7., -2., 4., 3., 5.])
>>> v = np.array([1., 3., -5.])
>>> q_up, r_up = linalg.qr_update(q, r, u, v, False)
>>> q_up
array([[ 0.54073807,  0.18645997,  0.81707661, -0.02136616,  0.06902409],  # may
↪vary (signs)
       [ 0.21629523, -0.63257324,  0.06567893,  0.34125904, -0.65749222],
       [ 0.05407381,  0.64757787, -0.12781284, -0.20031219, -0.72198188],
       [ 0.48666426, -0.30466718, -0.27487277, -0.77079214,  0.0256951 ],
       [ 0.64888568,  0.23001   , -0.4859845 ,  0.49883891,  0.20253783]])
>>> r_up
array([[ 18.49324201,  24.11691794, -44.98940746],  # may vary (signs)
       [  0.        ,  31.95894662, -27.40998201],
       [  0.        ,   0.        ,  -9.25451794],
       [  0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ]])
```

The update is equivalent, but faster than the following.

```
>>> a_up = a + np.outer(u, v)
>>> q_direct, r_direct = linalg.qr(a_up)
```

Check that we have equivalent results:

```
>>> np.allclose(np.dot(q_up, r_up), a_up)
True
```

And the updated Q is still unitary:

```
>>> np.allclose(np.dot(q_up.T, q_up), np.eye(5))
True
```

Updating economic (reduced, thin) decompositions is also possible:

```
>>> qe, re = linalg.qr(a, mode='economic')
>>> qe_up, re_up = linalg.qr_update(qe, re, u, v, False)
>>> qe_up
array([[ 0.54073807,  0.18645997,  0.81707661],  # may vary (signs)
       [ 0.21629523, -0.63257324,  0.06567893],
       [ 0.05407381,  0.64757787, -0.12781284],
       [ 0.48666426, -0.30466718, -0.27487277],
       [ 0.64888568,  0.23001   , -0.4859845 ]])
>>> re_up
array([[ 18.49324201,  24.11691794, -44.98940746],  # may vary (signs)
       [  0.        ,  31.95894662, -27.40998201],
       [  0.        ,   0.        ,  -9.25451794]])
>>> np.allclose(np.dot(qe_up, re_up), a_up)
True
>>> np.allclose(np.dot(qe_up.T, qe_up), np.eye(3))
True
```

Similarly to the above, perform a rank 2 update.

```
>>> u2 = np.array([[ 7., -1,],
...                [-2.,  4.],
...                [ 4.,  2.],
...                [ 3., -6.],
...                [ 5.,  3.]])
>>> v2 = np.array([[ 1., 2.],
...                [ 3., 4.],
...                [-5., 2]])
>>> q_up2, r_up2 = linalg.qr_update(q, r, u2, v2, False)
>>> q_up2
array([[-0.33626508, -0.03477253,  0.61956287, -0.64352987, -0.29618884],  # may␣
↪vary (signs)
       [-0.50439762,  0.58319694, -0.43010077, -0.33395279,  0.33008064],
       [-0.21016568, -0.63123106,  0.0582249 , -0.13675572,  0.73163206],
       [ 0.12609941,  0.49694436,  0.64590024,  0.31191919,  0.47187344],
       [-0.75659643, -0.11517748,  0.10284903,  0.5986227 , -0.21299983]])
>>> r_up2
array([[-23.79075451, -41.1084062 ,  24.71548348],  # may vary (signs)
       [  0.        , -33.83931057,  11.02226551],
       [  0.        ,   0.        ,  48.91476811],
       [  0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ]])
```

This update is also a valid qr decomposition of `A + U V**T`.

```
>>> a_up2 = a + np.dot(u2, v2.T)
>>> np.allclose(a_up2, np.dot(q_up2, r_up2))
True
>>> np.allclose(np.dot(q_up2.T, q_up2), np.eye(5))
True
```

`scipy.linalg.`**`qr_delete`**(*Q*, *R*, *k*, *int p=1*, *which='row'*, *overwrite_qr=False*, *check_finite=True*)

QR downdate on row or column deletions

If `A = Q R` is the QR factorization of `A`, return the QR factorization of `A` where `p` rows or columns have been removed starting at row or column `k`.

Parameters **Q** : (M, M) or (M, N) array_like

Unitary/orthogonal matrix from QR decomposition.

R : (M, N) or (N, N) array_like
    Upper triangular matrix from QR decomposition.

k : int
    Index of the first row or column to delete.

p : int, optional
    Number of rows or columns to delete, defaults to 1.

which: {'row', 'col'}, optional
    Determines if rows or columns will be deleted, defaults to 'row'

overwrite_qr : bool, optional
    If True, consume Q and R, overwriting their contents with their downdated versions, and returning appropriately sized views. Defaults to False.

check_finite : bool, optional
    Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is True.

Returns

Q1 : ndarray
    Updated unitary/orthogonal factor

R1 : ndarray
    Updated upper triangular factor

See also:

qr, *qr_multiply*, *qr_insert*, *qr_update*

Notes

This routine does not guarantee that the diagonal entries of `R1` are positive.

New in version 0.16.0.

References

[R115], [R116], [R117]

Examples

```
>>> from scipy import linalg
>>> a = np.array([[  3.,  -2.,  -2.],
...               [  6.,  -9.,  -3.],
...               [ -3.,  10.,   1.],
...               [  6.,  -7.,   4.],
...               [  7.,   8.,  -6.]])
>>> q, r = linalg.qr(a)
```

Given this QR decomposition, update q and r when 2 rows are removed.

```
>>> q1, r1 = linalg.qr_delete(q, r, 2, 2, 'row', False)
>>> q1
array([[ 0.30942637,  0.15347579,  0.93845645],  # may vary (signs)
       [ 0.61885275,  0.71680171, -0.32127338],
       [ 0.72199487, -0.68017681, -0.12681844]])
>>> r1
array([[  9.69535971,  -0.4125685 ,  -6.80738023],  # may vary (signs)
       [  0.        , -12.19958144,   1.62370412],
       [  0.        ,   0.        ,  -0.15218213]])
```

The update is equivalent, but faster than the following.

```
>>> a1 = np.delete(a, slice(2,4), 0)
>>> a1
array([[ 3., -2., -2.],
       [ 6., -9., -3.],
       [ 7.,  8., -6.]])
>>> q_direct, r_direct = linalg.qr(a1)
```

Check that we have equivalent results:

```
>>> np.dot(q1, r1)
array([[ 3., -2., -2.],
       [ 6., -9., -3.],
       [ 7.,  8., -6.]])
>>> np.allclose(np.dot(q1, r1), a1)
True
```

And the updated Q is still unitary:

```
>>> np.allclose(np.dot(q1.T, q1), np.eye(3))
True
```

scipy.linalg.**qr_insert**(*Q, R, u, k, which='row', rcond=None, overwrite_qru=False, check_finite=True*)

QR update on row or column insertions

If `A = Q R` is the QR factorization of `A`, return the QR factorization of `A` where rows or columns have been inserted starting at row or column `k`.

Parameters

Q : (M, M) array_like
: Unitary/orthogonal matrix from the QR decomposition of A.

R : (M, N) array_like
: Upper triangular matrix from the QR decomposition of A.

u : (N,), (p, N), (M,), or (M, p) array_like
: Rows or columns to insert

k : int
: Index before which *u* is to be inserted.

which: {'row', 'col'}, optional
: Determines if rows or columns will be inserted, defaults to 'row'

rcond : float
: Lower bound on the reciprocal condition number of `Q` augmented with `u/||u||` Only used when updating economic mode (thin, (M,N) (N,N)) decompositions. If None, machine precision is used. Defaults to None.

overwrite_qru : bool, optional
: If True, consume Q, R, and u, if possible, while performing the update, otherwise make copies as necessary. Defaults to False.

check_finite : bool, optional
: Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default is True.

Returns

Q1 : ndarray
: Updated unitary/orthogonal factor

R1 : ndarray
: Updated upper triangular factor

Raises

LinAlgError :
: If updating a (M,N) (N,N) factorization and the reciprocal condition number of Q augmented with u/||u|| is smaller than rcond.

See also:

qr, *qr_multiply*, *qr_delete*, *qr_update*

Notes

This routine does not guarantee that the diagonal entries of R1 are positive.

New in version 0.16.0.

References

[R118], [R119], [R120]

Examples

```
>>> from scipy import linalg
>>> a = np.array([[  3.,  -2.,  -2.],
...               [  6.,  -7.,   4.],
...               [  7.,   8.,  -6.]])
>>> q, r = linalg.qr(a)
```

Given this QR decomposition, update q and r when 2 rows are inserted.

```
>>> u = np.array([[  6.,  -9.,  -3.],
...               [ -3.,  10.,   1.]])
>>> q1, r1 = linalg.qr_insert(q, r, u, 2, 'row')
>>> q1
array([[-0.25445668,  0.02246245,  0.18146236, -0.72798806,  0.60979671],  # may␣
↪vary (signs)
       [-0.50891336,  0.23226178, -0.82836478, -0.02837033, -0.00828114],
       [-0.50891336,  0.35715302,  0.38937158,  0.58110733,  0.35235345],
       [ 0.25445668, -0.52202743, -0.32165498,  0.36263239,  0.65404509],
       [-0.59373225, -0.73856549,  0.16065817, -0.0063658 , -0.27595554]])
>>> r1
array([[-11.78982612,   6.44623587,   3.81685018],  # may vary (signs)
       [  0.        , -16.01393278,   3.72202865],
       [  0.        ,   0.        ,  -6.13010256],
       [  0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ]])
```

The update is equivalent, but faster than the following.

```
>>> a1 = np.insert(a, 2, u, 0)
>>> a1
array([[  3.,  -2.,  -2.],
       [  6.,  -7.,   4.],
       [  6.,  -9.,  -3.],
       [ -3.,  10.,   1.],
       [  7.,   8.,  -6.]])
>>> q_direct, r_direct = linalg.qr(a1)
```

Check that we have equivalent results:

```
>>> np.dot(q1, r1)
array([[  3.,  -2.,  -2.],
       [  6.,  -7.,   4.],
       [  6.,  -9.,  -3.],
       [ -3.,  10.,   1.],
       [  7.,   8.,  -6.]])
```

```
>>> np.allclose(np.dot(q1, r1), a1)
True
```

And the updated Q is still unitary:

```
>>> np.allclose(np.dot(q1.T, q1), np.eye(5))
True
```

`scipy.linalg.rq`(*a*, *overwrite_a=False*, *lwork=None*, *mode='full'*, *check_finite=True*)

Compute RQ decomposition of a matrix.

Calculate the decomposition `A = R Q` where Q is unitary/orthogonal and R upper triangular.

Parameters

a : (M, N) array_like
  Matrix to be decomposed

overwrite_a : bool, optional
  Whether data in a is overwritten (may improve performance)

lwork : int, optional
  Work array size, lwork >= a.shape[1]. If None or -1, an optimal size is computed.

mode : {'full', 'r', 'economic'}, optional
  Determines what information is to be returned: either both Q and R ('full', default), only R ('r') or both Q and R but computed in economy-size ('economic', see Notes).

check_finite : bool, optional
  Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

R : float or complex ndarray
  Of shape (M, N) or (M, K) for `mode='economic'`. `K = min(M, N)`.

Q : float or complex ndarray
  Of shape (N, N) or (K, N) for `mode='economic'`. Not returned if `mode='r'`.

Raises

LinAlgError
  If decomposition fails.

Notes

This is an interface to the LAPACK routines sgerqf, dgerqf, cgerqf, zgerqf, sorgrq, dorgrq, cungrq and zungrq.

If `mode=economic`, the shapes of Q and R are (K, N) and (M, K) instead of (N,N) and (M,N), with `K=min(M, N)`.

Examples

```
>>> from scipy import linalg
>>> from numpy import random, dot, allclose
>>> a = random.randn(6, 9)
>>> r, q = linalg.rq(a)
>>> allclose(a, dot(r, q))
True
>>> r.shape, q.shape
((6, 9), (9, 9))
>>> r2 = linalg.rq(a, mode='r')
>>> allclose(r, r2)
True
>>> r3, q3 = linalg.rq(a, mode='economic')
>>> r3.shape, q3.shape
((6, 6), (6, 9))
```

`scipy.linalg.qz(A, B, output='real', lwork=None, sort=None, overwrite_a=False, overwrite_b=False, check_finite=True)`

QZ decomposition for generalized eigenvalues of a pair of matrices.

The QZ, or generalized Schur, decomposition for a pair of N x N nonsymmetric matrices (A,B) is:

```
(A,B) = (Q*AA*Z', Q*BB*Z')
```

where AA, BB is in generalized Schur form if BB is upper-triangular with non-negative diagonal and AA is upper-triangular, or for real QZ decomposition (`output='real'`) block upper triangular with 1x1 and 2x2 blocks. In this case, the 1x1 blocks correspond to real generalized eigenvalues and 2x2 blocks are 'standardized' by making the corresponding elements of BB have the form:

```
[ a 0 ]
[ 0 b ]
```

and the pair of corresponding 2x2 blocks in AA and BB will have a complex conjugate pair of generalized eigenvalues. If (`output='complex'`) or A and B are complex matrices, Z' denotes the conjugate-transpose of Z. Q and Z are unitary matrices.

Parameters

A : (N, N) array_like
: 2d array to decompose

B : (N, N) array_like
: 2d array to decompose

output : {'real', 'complex'}, optional
: Construct the real or complex QZ decomposition for real matrices. Default is 'real'.

lwork : int, optional
: Work array size. If None or -1, it is automatically computed.

sort : {None, callable, 'lhp', 'rhp', 'iuc', 'ouc'}, optional
: NOTE: THIS INPUT IS DISABLED FOR NOW. Use ordqz instead.
Specifies whether the upper eigenvalues should be sorted. A callable may be passed that, given a eigenvalue, returns a boolean denoting whether the eigenvalue should be sorted to the top-left (True). For real matrix pairs, the sort function takes three real arguments (alphar, alphai, beta). The eigenvalue `x = (alphar + alphai*1j)/beta`. For complex matrix pairs or output='complex', the sort function takes two complex arguments (alpha, beta). The eigenvalue `x = (alpha/beta)`. Alternatively, string parameters may be used:
  - 'lhp' Left-hand plane (x.real < 0.0)
  - 'rhp' Right-hand plane (x.real > 0.0)
  - 'iuc' Inside the unit circle (x*x.conjugate() < 1.0)
  - 'ouc' Outside the unit circle (x*x.conjugate() > 1.0)

  Defaults to None (no sorting).

overwrite_a : bool, optional
: Whether to overwrite data in a (may improve performance)

overwrite_b : bool, optional
: Whether to overwrite data in b (may improve performance)

check_finite : bool, optional
: If true checks the elements of *A* and *B* are finite numbers. If false does no checking and passes matrix through to underlying algorithm.

Returns

AA : (N, N) ndarray
: Generalized Schur form of A.

BB : (N, N) ndarray
: Generalized Schur form of B.

Q : (N, N) ndarray
: The left Schur vectors.

Z : (N, N) ndarray

The right Schur vectors.

See also:

ordqz

Notes

Q is transposed versus the equivalent function in Matlab.

New in version 0.11.0.

Examples

```
>>> from scipy import linalg
>>> np.random.seed(1234)
>>> A = np.arange(9).reshape((3, 3))
>>> B = np.random.randn(3, 3)
```

```
>>> AA, BB, Q, Z = linalg.qz(A, B)
>>> AA
array([[-13.40928183,  -4.62471562,   1.09215523],
       [  0.        ,   0.        ,   1.22805978],
       [  0.        ,   0.        ,   0.31973817]])
>>> BB
array([[ 0.33362547, -1.37393632,  0.02179805],
       [ 0.        ,  1.68144922,  0.74683866],
       [ 0.        ,  0.        ,  0.9258294 ]])
>>> Q
array([[ 0.14134727, -0.97562773,  0.16784365],
       [ 0.49835904, -0.07636948, -0.86360059],
       [ 0.85537081,  0.20571399,  0.47541828]])
>>> Z
array([[-0.24900855, -0.51772687,  0.81850696],
       [-0.79813178,  0.58842606,  0.12938478],
       [-0.54861681, -0.6210585 , -0.55973739]])
```

scipy.linalg.**ordqz**(*A*, *B*, *sort='lhp'*, *output='real'*, *overwrite_a=False*, *overwrite_b=False*, *check_finite=True*)

QZ decomposition for a pair of matrices with reordering.

New in version 0.17.0.

Parameters **A** : (N, N) array_like

2d array to decompose

B : (N, N) array_like

2d array to decompose

sort : {callable, 'lhp', 'rhp', 'iuc', 'ouc'}, optional

Specifies whether the upper eigenvalues should be sorted. A callable may be passed that, given an ordered pair `(alpha, beta)` representing the eigenvalue `x = (alpha/beta)`, returns a boolean denoting whether the eigenvalue should be sorted to the top-left (True). For the real matrix pairs `beta` is real while `alpha` can be complex, and for complex matrix pairs both `alpha` and `beta` can be complex. The callable must be able to accept a numpy array. Alternatively, string parameters may be used:

- 'lhp' Left-hand plane (x.real < 0.0)
- 'rhp' Right-hand plane (x.real > 0.0)
- 'iuc' Inside the unit circle (x*x.conjugate() < 1.0)
- 'ouc' Outside the unit circle (x*x.conjugate() > 1.0)

With the predefined sorting functions, an infinite eigenvalue (i.e. `alpha != 0` and `beta = 0`) is considered to lie in neither the left-hand nor the right-hand plane, but it is considered to lie outside the unit circle. For the eigenvalue `(alpha, beta) = (0, 0)` the predefined sorting functions all return *False*.

output : str {'real','complex'}, optional
Construct the real or complex QZ decomposition for real matrices. Default is 'real'.

overwrite_a : bool, optional
If True, the contents of A are overwritten.

overwrite_b : bool, optional
If True, the contents of B are overwritten.

check_finite : bool, optional
If true checks the elements of *A* and *B* are finite numbers. If false does no checking and passes matrix through to underlying algorithm.

Returns

AA : (N, N) ndarray
Generalized Schur form of A.

BB : (N, N) ndarray
Generalized Schur form of B.

alpha : (N,) ndarray
alpha = alphar + alphai * 1j. See notes.

beta : (N,) ndarray
See notes.

Q : (N, N) ndarray
The left Schur vectors.

Z : (N, N) ndarray
The right Schur vectors.

See also:

qz

Notes

On exit, `(ALPHAR(j) + ALPHAI(j)*i)/BETA(j), j=1,...,N`, will be the generalized eigenvalues. `ALPHAR(j) + ALPHAI(j)*i` and `BETA(j),j=1,...,N` are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A,B) were further reduced to triangular form using complex unitary transformations. If ALPHAI(j) is zero, then the j-th eigenvalue is real; if positive, then the `j`-th and `(j+1)`-st eigenvalues are a complex conjugate pair, with `ALPHAI(j+1)` negative.

`scipy.linalg.`**`schur`**(*a*, *output='real'*, *lwork=None*, *overwrite_a=False*, *sort=None*, *check_finite=True*)

Compute Schur decomposition of a matrix.

The Schur decomposition is:

```
A = Z T Z^H
```

where Z is unitary and T is either upper-triangular, or for real Schur decomposition (output='real'), quasi-upper triangular. In the quasi-triangular form, 2x2 blocks describing complex-valued eigenvalue pairs may extrude from the diagonal.

Parameters

a : (M, M) array_like
Matrix to decompose

output : {'real', 'complex'}, optional
Construct the real or complex Schur decomposition (for real matrices).

lwork : int, optional
Work array size. If None or -1, it is automatically computed.

overwrite_a : bool, optional
Whether to overwrite data in a (may improve performance).

sort : {None, callable, 'lhp', 'rhp', 'iuc', 'ouc'}, optional
Specifies whether the upper eigenvalues should be sorted. A callable may be passed that, given a eigenvalue, returns a boolean denoting whether the eigenvalue should be sorted to the top-left (True). Alternatively, string parameters may be used:

```
'lhp'   Left-hand plane (x.real < 0.0)
'rhp'   Right-hand plane (x.real > 0.0)
'iuc'   Inside the unit circle (x*x.conjugate() <= 1.0)
'ouc'   Outside the unit circle (x*x.conjugate() > 1.0)
```

Defaults to None (no sorting).

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

T : (M, M) ndarray
Schur form of A. It is real-valued for the real Schur decomposition.

Z : (M, M) ndarray
An unitary Schur transformation matrix for A. It is real-valued for the real Schur decomposition.

sdim : int
If and only if sorting was requested, a third return value will contain the number of eigenvalues satisfying the sort condition.

Raises

LinAlgError
Error raised under three conditions:

1. The algorithm failed due to a failure of the QR algorithm to compute all eigenvalues
2. If eigenvalue sorting was requested, the eigenvalues could not be reordered due to a failure to separate eigenvalues, usually because of poor conditioning
3. If eigenvalue sorting was requested, roundoff errors caused the leading eigenvalues to no longer satisfy the sorting condition

See also:

`rsf2csf` Convert real Schur form to complex Schur form

`scipy.linalg.`**`rsf2csf`**(*T*, *Z*, *check_finite=True*)
Convert real Schur form to complex Schur form.

Convert a quasi-diagonal real-valued Schur form to the upper triangular complex-valued Schur form.

Parameters

T : (M, M) array_like
Real Schur form of the original matrix

Z : (M, M) array_like
Schur transformation matrix

check_finite : bool, optional
Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

T : (M, M) ndarray
Complex Schur form of the original matrix

Z : (M, M) ndarray
Schur transformation matrix corresponding to the complex form

See also:

schur	Schur decompose a matrix

`scipy.linalg.hessenberg(a, calc_q=False, overwrite_a=False, check_finite=True)`

Compute Hessenberg form of a matrix.

The Hessenberg decomposition is:

```
A = Q H Q^H
```

where *Q* is unitary/orthogonal and *H* has only zero elements below the first sub-diagonal.

Parameters

a : (M, M) array_like
Matrix to bring into Hessenberg form.

calc_q : bool, optional
Whether to compute the transformation matrix. Default is False.

overwrite_a : bool, optional
Whether to overwrite *a*; may improve performance. Default is False.

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

H : (M, M) ndarray
Hessenberg form of *a*.

Q : (M, M) ndarray
Unitary/orthogonal similarity transformation matrix `A = Q H Q^H`. Only returned if `calc_q=True`.

See also:

scipy.linalg.interpolative – Interpolative matrix decompositions

5.9.4 Matrix Functions

expm(A[, q])	Compute the matrix exponential using Pade approximation.
logm(A[, disp])	Compute matrix logarithm.
cosm(A)	Compute the matrix cosine.
sinm(A)	Compute the matrix sine.
tanm(A)	Compute the matrix tangent.
coshm(A)	Compute the hyperbolic matrix cosine.
sinhm(A)	Compute the hyperbolic matrix sine.
tanhm(A)	Compute the hyperbolic matrix tangent.
signm(A[, disp])	Matrix sign function.
sqrtm(A[, disp, blocksize])	Matrix square root.
funm(A, func[, disp])	Evaluate a matrix function specified by a callable.
expm_frechet(A, E[, method, compute_expm, ...])	Frechet derivative of the matrix exponential of A in the direction E.
expm_cond(A[, check_finite])	Relative condition number of the matrix exponential in the Frobenius norm.
fractional_matrix_power(A, t)	Compute the fractional power of a matrix.

`scipy.linalg.expm(A, q=None)`

Compute the matrix exponential using Pade approximation.

Parameters

A : (N, N) array_like or sparse matrix

Matrix to be exponentiated.

Returns **expm** : (N, N) ndarray
Matrix exponential of *A*.

References

[R97]

Examples

```
>>> from scipy.linalg import expm, sinm, cosm
```

Matrix version of the formula exp(0) = 1:

```
>>> expm(np.zeros((2,2)))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Euler's identity (exp(i*theta) = cos(theta) + i*sin(theta)) applied to a matrix:

```
>>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
>>> expm(1j*a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
>>> cosm(a) + 1j*sinm(a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
```

`scipy.linalg.`**`logm`**(*A*, *disp=True*)

Compute matrix logarithm.

The matrix logarithm is the inverse of expm: expm(logm(*A*)) == *A*

Parameters **A** : (N, N) array_like
Matrix whose logarithm to evaluate
disp : bool, optional
Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns **logm** : (N, N) ndarray
Matrix logarithm of *A*
errest : float
(if disp == False)
1-norm of the estimated error, ||err||_1 / ||A||_1

References

[R105], *[R106]*, *[R107]*

Examples

```
>>> from scipy.linalg import logm, expm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> b = logm(a)
>>> b
array([[-1.02571087,  2.05142174],
       [ 0.68380725,  1.02571087]])
>>> expm(b)         # Verify expm(logm(a)) returns a
array([[ 1.,  3.],
       [ 1.,  4.]])
```

`scipy.linalg.cosm(A)`

Compute the matrix cosine.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array

Returns **cosm** : (N, N) ndarray
Matrix cosine of A

Examples

```
>>> from scipy.linalg import expm, sinm, cosm
```

Euler's identity (exp(i*theta) = cos(theta) + i*sin(theta)) applied to a matrix:

```
>>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
>>> expm(1j*a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
>>> cosm(a) + 1j*sinm(a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
```

`scipy.linalg.sinm(A)`

Compute the matrix sine.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array.

Returns **sinm** : (N, N) ndarray
Matrix sine of *A*

Examples

```
>>> from scipy.linalg import expm, sinm, cosm
```

Euler's identity (exp(i*theta) = cos(theta) + i*sin(theta)) applied to a matrix:

```
>>> a = np.array([[1.0, 2.0], [-1.0, 3.0]])
>>> expm(1j*a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
>>> cosm(a) + 1j*sinm(a)
array([[ 0.42645930+1.89217551j, -2.13721484-0.97811252j],
       [ 1.06860742+0.48905626j, -1.71075555+0.91406299j]])
```

`scipy.linalg.tanm(A)`

Compute the matrix tangent.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array.

Returns **tanm** : (N, N) ndarray
Matrix tangent of *A*

Examples

```
>>> from scipy.linalg import tanm, sinm, cosm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> t = tanm(a)
>>> t
array([[ -2.00876993,  -8.41880636],
       [ -2.80626879, -10.42757629]])
```

Verify tanm(a) = sinm(a).dot(inv(cosm(a)))

```
>>> s = sinm(a)
>>> c = cosm(a)
>>> s.dot(np.linalg.inv(c))
array([[ -2.00876993,  -8.41880636],
       [ -2.80626879, -10.42757629]])
```

scipy.linalg.**coshm**(*A*)

Compute the hyperbolic matrix cosine.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array.

Returns **coshm** : (N, N) ndarray
Hyperbolic matrix cosine of *A*

Examples

```
>>> from scipy.linalg import tanhm, sinhm, coshm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> c = coshm(a)
>>> c
array([[ 11.24592233,  38.76236492],
       [ 12.92078831,  50.00828725]])
```

Verify tanhm(a) = sinhm(a).dot(inv(coshm(a)))

```
>>> t = tanhm(a)
>>> s = sinhm(a)
>>> t - s.dot(np.linalg.inv(c))
array([[  2.72004641e-15,   4.55191440e-15],
       [  0.00000000e+00,  -5.55111512e-16]])
```

scipy.linalg.**sinhm**(*A*)

Compute the hyperbolic matrix sine.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array.

Returns **sinhm** : (N, N) ndarray
Hyperbolic matrix sine of *A*

Examples

```
>>> from scipy.linalg import tanhm, sinhm, coshm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> s = sinhm(a)
```

```
>>> s
array([[ 10.57300653,  39.28826594],
       [ 13.09608865,  49.86127247]])
```

Verify tanhm(a) = sinhm(a).dot(inv(coshm(a)))

```
>>> t = tanhm(a)
>>> c = coshm(a)
>>> t - s.dot(np.linalg.inv(c))
array([[  2.72004641e-15,   4.55191440e-15],
       [  0.00000000e+00,  -5.55111512e-16]])
```

scipy.linalg.**tanhm**(*A*)

Compute the hyperbolic matrix tangent.

This routine uses expm to compute the matrix exponentials.

Parameters **A** : (N, N) array_like
Input array

Returns **tanhm** : (N, N) ndarray
Hyperbolic matrix tangent of *A*

Examples

```
>>> from scipy.linalg import tanhm, sinhm, coshm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> t = tanhm(a)
>>> t
array([[ 0.3428582 ,  0.51987926],
       [ 0.17329309,  0.86273746]])
```

Verify tanhm(a) = sinhm(a).dot(inv(coshm(a)))

```
>>> s = sinhm(a)
>>> c = coshm(a)
>>> t - s.dot(np.linalg.inv(c))
array([[  2.72004641e-15,   4.55191440e-15],
       [  0.00000000e+00,  -5.55111512e-16]])
```

scipy.linalg.**signm**(*A*, *disp=True*)

Matrix sign function.

Extension of the scalar sign(x) to matrices.

Parameters **A** : (N, N) array_like
Matrix at which to evaluate the sign function

disp : bool, optional
Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns **signm** : (N, N) ndarray
Value of the sign function at *A*

errest : float
(if disp == False)
1-norm of the estimated error, ||err||_1 / ||A||_1

Examples

```
>>> from scipy.linalg import signm, eigvals
>>> a = [[1,2,3], [1,2,1], [1,1,1]]
>>> eigvals(a)
array([ 4.12488542+0.j, -0.76155718+0.j,  0.63667176+0.j])
>>> eigvals(signm(a))
array([-1.+0.j,  1.+0.j,  1.+0.j])
```

`scipy.linalg.sqrtm(A, disp=True, blocksize=64)`

Matrix square root.

Parameters

A : (N, N) array_like
Matrix whose square root to evaluate

disp : bool, optional
Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

blocksize : integer, optional
If the blocksize is not degenerate with respect to the size of the input array, then use a blocked algorithm. (Default: 64)

Returns

sqrtm : (N, N) ndarray
Value of the sqrt function at *A*

errest : float
(if disp == False)
Frobenius norm of the estimated error, ||err||_F / ||A||_F

References

[R132]

Examples

```
>>> from scipy.linalg import sqrtm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> r = sqrtm(a)
>>> r
array([[ 0.75592895,  1.13389342],
       [ 0.37796447,  1.88982237]])
>>> r.dot(r)
array([[ 1.,  3.],
       [ 1.,  4.]])
```

`scipy.linalg.funm(A, func, disp=True)`

Evaluate a matrix function specified by a callable.

Returns the value of matrix-valued function `f` at *A*. The function `f` is an extension of the scalar-valued function *func* to matrices.

Parameters

A : (N, N) array_like
Matrix at which to evaluate the function

func : callable
Callable object that evaluates a scalar function f. Must be vectorized (eg. using vectorize).

disp : bool, optional
Print warning if error in the result is estimated large instead of returning estimated error. (Default: True)

Returns

funm : (N, N) ndarray
Value of the matrix function specified by func evaluated at *A*

errest : float
(if disp == False)
1-norm of the estimated error, ||err||_1 / ||A||_1

Notes

This function implements the general algorithm based on Schur decomposition (Algorithm 9.1.1. in *[R100]*).

If the input matrix is known to be diagonalizable, then relying on the eigendecomposition is likely to be faster. For example, if your matrix is Hermitian, you can do

```
>>> from scipy.linalg import eigh
>>> def funm_herm(a, func, check_finite=False):
...     w, v = eigh(a, check_finite=check_finite)
...     ## if you further know that your matrix is positive semidefinite,
...     ## you can optionally guard against precision errors by doing
...     # w = np.maximum(w, 0)
...     w = func(w)
...     return (v * w).dot(v.conj().T)
```

References

[R100]

Examples

```
>>> from scipy.linalg import funm
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> funm(a, lambda x: x*x)
array([[  4.,  15.],
       [  5.,  19.]])
>>> a.dot(a)
array([[  4.,  15.],
       [  5.,  19.]])
```

`scipy.linalg.`**`expm_frechet`**(*A*, *E*, *method=None*, *compute_expm=True*, *check_finite=True*)

Frechet derivative of the matrix exponential of A in the direction E.

Parameters

A : (N, N) array_like
Matrix of which to take the matrix exponential.

E : (N, N) array_like
Matrix direction in which to take the Frechet derivative.

method : str, optional
Choice of algorithm. Should be one of
- *SPS* (default)
- *blockEnlarge*

compute_expm : bool, optional
Whether to compute also *expm_A* in addition to *expm_frechet_AE*. Default is True.

check_finite : bool, optional
Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns

expm_A : ndarray
Matrix exponential of A.

expm_frechet_AE : ndarray
Frechet derivative of the matrix exponential of A in the direction E.

For `compute_expm = False`, only *expm_frechet_AE* is returned.

See also:

expm Compute the exponential of a matrix.

Notes

This section describes the available implementations that can be selected by the *method* parameter. The default method is *SPS*.

Method *blockEnlarge* is a naive algorithm.

Method *SPS* is Scaling-Pade-Squaring *[R98]*. It is a sophisticated implementation which should take only about 3/8 as much time as the naive implementation. The asymptotics are the same.

New in version 0.13.0.

References

[R98]

Examples

```
>>> import scipy.linalg
>>> A = np.random.randn(3, 3)
>>> E = np.random.randn(3, 3)
>>> expm_A, expm_frechet_AE = scipy.linalg.expm_frechet(A, E)
>>> expm_A.shape, expm_frechet_AE.shape
((3, 3), (3, 3))
```

```
>>> import scipy.linalg
>>> A = np.random.randn(3, 3)
>>> E = np.random.randn(3, 3)
>>> expm_A, expm_frechet_AE = scipy.linalg.expm_frechet(A, E)
>>> M = np.zeros((6, 6))
>>> M[:3, :3] = A; M[:3, 3:] = E; M[3:, 3:] = A
>>> expm_M = scipy.linalg.expm(M)
>>> np.allclose(expm_A, expm_M[:3, :3])
True
>>> np.allclose(expm_frechet_AE, expm_M[:3, 3:])
True
```

scipy.linalg.**expm_cond**(*A*, *check_finite=True*)

Relative condition number of the matrix exponential in the Frobenius norm.

Parameters **A** : 2d array_like

Square input matrix with shape (N, N).

check_finite : bool, optional

Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.

Returns **kappa** : float

The relative condition number of the matrix exponential in the Frobenius norm

See also:

expm Compute the exponential of a matrix.

expm_frechet

Compute the Frechet derivative of the matrix exponential.

Notes

A faster estimate for the condition number in the 1-norm has been published but is not yet implemented in scipy.

New in version 0.14.0.

`scipy.linalg.`**`fractional_matrix_power`**(*A*, *t*)

Compute the fractional power of a matrix.

Proceeds according to the discussion in section (6) of *[R99]*.

Parameters

A : (N, N) array_like
Matrix whose fractional power to evaluate.

t : float
Fractional power.

Returns

X : (N, N) array_like
The fractional power of the matrix.

References

[R99]

Examples

```
>>> from scipy.linalg import fractional_matrix_power
>>> a = np.array([[1.0, 3.0], [1.0, 4.0]])
>>> b = fractional_matrix_power(a, 0.5)
>>> b
array([[ 0.75592895,  1.13389342],
       [ 0.37796447,  1.88982237]])
>>> np.dot(b, b)      # Verify square root
array([[ 1.,  3.],
       [ 1.,  4.]])
```

5.9.5 Matrix Equation Solvers

solve_sylvester(a, b, q)	Computes a solution (X) to the Sylvester equation $AX + XB = Q$.
solve_continuous_are(a, b, q, r[, e, s, ...])	Solves the continuous-time algebraic Riccati equation (CARE).
solve_discrete_are(a, b, q, r[, e, s, balanced])	Solves the discrete-time algebraic Riccati equation (DARE).
solve_discrete_lyapunov(a, q[, method])	Solves the discrete Lyapunov equation $AXA^H - X + Q = 0$.
solve_lyapunov(a, q)	Solves the continuous Lyapunov equation $AX + XA^H = Q$.

`scipy.linalg.`**`solve_sylvester`**(*a*, *b*, *q*)

Computes a solution (X) to the Sylvester equation $AX + XB = Q$.

Parameters

a : (M, M) array_like
Leading matrix of the Sylvester equation

b : (N, N) array_like
Trailing matrix of the Sylvester equation

q : (M, N) array_like
Right-hand side

Returns **x** : (M, N) ndarray
    The solution to the Sylvester equation.

Raises **LinAlgError**
    If solution was not found

Notes

Computes a solution to the Sylvester matrix equation via the Bartels- Stewart algorithm. The A and B matrices first undergo Schur decompositions. The resulting matrices are used to construct an alternative Sylvester equation (`RY + YS^T = F`) where the R and S matrices are in quasi-triangular form (or, when R, S or F are complex, triangular form). The simplified equation is then solved using `*TRSYL` from LAPACK directly.

New in version 0.11.0.

`scipy.linalg.`**`solve_continuous_are`**(*a*, *b*, *q*, *r*, *e=None*, *s=None*, *balanced=True*)

Solves the continuous-time algebraic Riccati equation (CARE).

The CARE is defined as

$$XA + A^H X - XBR^{-1}B^H X + Q = 0$$

The limitations for a solution to exist are :

- All eigenvalues of A on the right half plane, should be controllable.
- The associated hamiltonian pencil (See Notes), should have eigenvalues sufficiently away from the imaginary axis.

Moreover, if `e` or `s` is not precisely `None`, then the generalized version of CARE

$$E^H XA + A^H XE - (E^H XB + S)R^{-1}(B^H XE + S^H) + Q = 0$$

is solved. When omitted, `e` is assumed to be the identity and `s` is assumed to be the zero matrix with sizes compatible with `a` and `b` respectively.

Parameters **a** : (M, M) array_like
    Square matrix

b : (M, N) array_like
    Input

q : (M, M) array_like
    Input

r : (N, N) array_like
    Nonsingular square matrix

e : (M, M) array_like, optional
    Nonsingular square matrix

s : (M, N) array_like, optional
    Input

balanced : bool, optional
    The boolean that indicates whether a balancing step is performed on the data. The default is set to True.

Returns **x** : (M, M) ndarray
    Solution to the continuous-time algebraic Riccati equation.

Raises **LinAlgError**
    For cases where the stable subspace of the pencil could not be isolated. See Notes section and the references for details.

See also:

`solve_discrete_are`
    Solves the discrete-time algebraic Riccati equation

Notes

The equation is solved by forming the extended hamiltonian matrix pencil, as described in *[R124]*, $H - \lambda J$ given by the block matrices

```
[ A    0    B ]             [ E   0    0 ]
[-Q  -A^H  -S ] - \lambda * [ 0  E^H   0 ]
[ S^H B^H   R ]             [ 0   0    0 ]
```

and using a QZ decomposition method.

In this algorithm, the fail conditions are linked to the symmetry of the product $U_2 U_1^{-1}$ and condition number of U_1. Here, U is the 2m-by-m matrix that holds the eigenvectors spanning the stable subspace with 2m rows and partitioned into two m-row matrices. See *[R124]* and *[R125]* for more details.

In order to improve the QZ decomposition accuracy, the pencil goes through a balancing step where the sum of absolute values of H and J entries (after removing the diagonal entries of the sum) is balanced following the recipe given in *[R126]*.

New in version 0.11.0.

References

[R124], *[R125]*, *[R126]*

`scipy.linalg.`**`solve_discrete_are`**(*a*, *b*, *q*, *r*, *e=None*, *s=None*, *balanced=True*)

Solves the discrete-time algebraic Riccati equation (DARE).

The DARE is defined as

$$A^H X A - X - (A^H X B)(R + B^H X B)^{-1}(B^H X A) + Q = 0$$

The limitations for a solution to exist are :

- All eigenvalues of A outside the unit disc, should be controllable.
- The associated symplectic pencil (See Notes), should have eigenvalues sufficiently away from the unit circle.

Moreover, if `e` and `s` are not both precisely `None`, then the generalized version of DARE

$$A^H X A - E^H X E - (A^H X B + S)(R + B^H X B)^{-1}(B^H X A + S^H) + Q = 0$$

is solved. When omitted, `e` is assumed to be the identity and `s` is assumed to be the zero matrix.

Parameters

a : (M, M) array_like
    Square matrix

b : (M, N) array_like
    Input

q : (M, M) array_like
    Input

r : (N, N) array_like
    Square matrix

e : (M, M) array_like, optional
    Nonsingular square matrix

s : (M, N) array_like, optional
    Input

balanced : bool
    The boolean that indicates whether a balancing step is performed on the data. The default is set to True.

Returns **x** : (M, M) ndarray
Solution to the discrete algebraic Riccati equation.

Raises **LinAlgError**
For cases where the stable subspace of the pencil could not be isolated. See Notes section and the references for details.

See also:

solve_continuous_are
Solves the continuous algebraic Riccati equation

Notes

The equation is solved by forming the extended symplectic matrix pencil, as described in *[R127]*, $H - \lambda J$ given by the block matrices

```
[  A    0    B ]             [ E    0    B ]
[ -Q  E^H -S ] - \lambda * [ 0   A^H   0 ]
[ S^H   0    R ]             [ 0 -B^H   0 ]
```

and using a QZ decomposition method.

In this algorithm, the fail conditions are linked to the symmetry of the product $U_2 U_1^{-1}$ and condition number of U_1. Here, U is the 2m-by-m matrix that holds the eigenvectors spanning the stable subspace with 2m rows and partitioned into two m-row matrices. See *[R127]* and *[R128]* for more details.

In order to improve the QZ decomposition accuracy, the pencil goes through a balancing step where the sum of absolute values of H and J rows/cols (after removing the diagonal entries) is balanced following the recipe given in *[R129]*. If the data has small numerical noise, balancing may amplify their effects and some clean up is required.

New in version 0.11.0.

References

[R127], *[R128]*, *[R129]*

scipy.linalg.**solve_discrete_lyapunov**(*a*, *q*, *method=None*)
Solves the discrete Lyapunov equation $AXA^H - X + Q = 0$.

Parameters **a, q** : (M, M) array_like
Square matrices corresponding to A and Q in the equation above respectively. Must have the same shape.
method : {'direct', 'bilinear'}, optional
Type of solver.
If not given, chosen to be `direct` if `M` is less than 10 and `bilinear` otherwise.

Returns **x** : ndarray
Solution to the discrete Lyapunov equation

See also:

solve_lyapunov
computes the solution to the continuous Lyapunov equation

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *direct* if `M` is less than 10 and `bilinear` otherwise.

Method *direct* uses a direct analytical solution to the discrete Lyapunov equation. The algorithm is given in, for example, *[R130]*. However it requires the linear solution of a system with dimension M^2 so that performance degrades rapidly for even moderately sized matrices.

Method *bilinear* uses a bilinear transformation to convert the discrete Lyapunov equation to a continuous Lyapunov equation $(BX + XB' = -C)$ where $B = (A - I)(A + I)^{-1}$ and $C = 2(A' + I)^{-1}Q(A + I)^{-1}$. The continuous equation can be efficiently solved since it is a special case of a Sylvester equation. The transformation algorithm is from Popov (1964) as described in *[R131]*.

New in version 0.11.0.

References

[R130], *[R131]*

`scipy.linalg.`**`solve_lyapunov`**(*a*, *q*)

Solves the continuous Lyapunov equation $AX + XA^H = Q$.

Uses the Bartels-Stewart algorithm to find X.

Parameters **a** : array_like
A square matrix
q : array_like
Right-hand side square matrix

Returns **x** : array_like
Solution to the continuous Lyapunov equation

See also:

`solve_sylvester`
computes the solution to the Sylvester equation

Notes

Because the continuous Lyapunov equation is just a special form of the Sylvester equation, this solver relies entirely on solve_sylvester for a solution.

New in version 0.11.0.

5.9.6 Special Matrices

block_diag(*arrs)	Create a block diagonal matrix from provided arrays.
circulant(c)	Construct a circulant matrix.
companion(a)	Create a companion matrix.
dft(n[, scale])	Discrete Fourier transform matrix.
hadamard(n[, dtype])	Construct a Hadamard matrix.
hankel(c[, r])	Construct a Hankel matrix.
helmert(n[, full])	Create a Helmert matrix of order *n*.
hilbert(n)	Create a Hilbert matrix of order *n*.
invhilbert(n[, exact])	Compute the inverse of the Hilbert matrix of order *n*.
leslie(f, s)	Create a Leslie matrix.
pascal(n[, kind, exact])	Returns the n x n Pascal matrix.
invpascal(n[, kind, exact])	Returns the inverse of the n x n Pascal matrix.
toeplitz(c[, r])	Construct a Toeplitz matrix.
tri(N[, M, k, dtype])	Construct (N, M) matrix filled with ones at and below the k-th diagonal.

`scipy.linalg.`**`block_diag`**`(*arrs)`

Create a block diagonal matrix from provided arrays.

Given the inputs *A*, *B* and *C*, the output will have these arrays arranged on the diagonal:

```
[[A, 0, 0],
 [0, B, 0],
 [0, 0, C]]
```

Parameters **A, B, C, ...** : array_like, up to 2-D

Input arrays. A 1-D array or array_like sequence of length *n* is treated as a 2-D array with shape `(1,n)`.

Returns **D** : ndarray

Array with *A*, *B*, *C*, ... on the diagonal. *D* has the same dtype as *A*.

Notes

If all the input arrays are square, the output is known as a block diagonal matrix.

Empty sequences (i.e., array-likes of zero size) will not be ignored. Noteworthy, both [] and [[]] are treated as matrices with shape `(1,0)`.

Examples

```
>>> from scipy.linalg import block_diag
>>> A = [[1, 0],
...      [0, 1]]
>>> B = [[3, 4, 5],
...      [6, 7, 8]]
>>> C = [[7]]
>>> P = np.zeros((2, 0), dtype='int32')
>>> block_diag(A, B, C)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 3, 4, 5, 0],
       [0, 0, 6, 7, 8, 0],
       [0, 0, 0, 0, 0, 7]])
>>> block_diag(A, P, B, C)
array([[1, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 3, 4, 5, 0],
       [0, 0, 6, 7, 8, 0],
       [0, 0, 0, 0, 0, 7]])
>>> block_diag(1.0, [2, 3], [[4, 5], [6, 7]])
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  2.,  3.,  0.,  0.],
       [ 0.,  0.,  0.,  4.,  5.],
       [ 0.,  0.,  0.,  6.,  7.]])
```

`scipy.linalg.`**`circulant`**`(c)`

Construct a circulant matrix.

Parameters **c** : (N,) array_like

1-D array, the first column of the matrix.

Returns **A** : (N, N) ndarray

A circulant matrix whose first column is *c*.

See also:

toeplitz Toeplitz matrix

hankel Hankel matrix

Notes

New in version 0.8.0.

Examples

```
>>> from scipy.linalg import circulant
>>> circulant([1, 2, 3])
array([[1, 3, 2],
       [2, 1, 3],
       [3, 2, 1]])
```

scipy.linalg.**companion**(*a*)

Create a companion matrix.

Create the companion matrix *[R95]* associated with the polynomial whose coefficients are given in *a*.

Parameters
a : (N,) array_like
1-D array of polynomial coefficients. The length of *a* must be at least two, and `a[0]` must not be zero.

Returns
c : (N-1, N-1) ndarray
The first row of *c* is `-a[1:]/a[0]`, and the first sub-diagonal is all ones. The data-type of the array is the same as the data-type of `1.0*a[0]`.

Raises
ValueError
If any of the following are true: a) `a.ndim != 1`; b) `a.size < 2`; c) `a[0] == 0`.

Notes

New in version 0.8.0.

References

[R95]

Examples

```
>>> from scipy.linalg import companion
>>> companion([1, -10, 31, -30])
array([[ 10., -31.,  30.],
       [  1.,   0.,   0.],
       [  0.,   1.,   0.]])
```

scipy.linalg.**dft**(*n*, *scale=None*)

Discrete Fourier transform matrix.

Create the matrix that computes the discrete Fourier transform of a sequence *[R96]*. The n-th primitive root of unity used to generate the matrix is exp(-2*pi*i/n), where i = sqrt(-1).

Parameters
n : int
Size the matrix to create.
scale : str, optional

Must be None, 'sqrtn', or 'n'. If *scale* is 'sqrtn', the matrix is divided by *sqrt(n)*. If *scale* is 'n', the matrix is divided by *n*. If *scale* is None (the default), the matrix is not normalized, and the return value is simply the Vandermonde matrix of the roots of unity.

Returns **m** : (n, n) ndarray
The DFT matrix.

Notes

When *scale* is None, multiplying a vector by the matrix returned by `dft` is mathematically equivalent to (but much less efficient than) the calculation performed by `scipy.fftpack.fft`.

New in version 0.14.0.

References

[R96]

Examples

```
>>> from scipy.linalg import dft
>>> np.set_printoptions(precision=5, suppress=True)
>>> x = np.array([1, 2, 3, 0, 3, 2, 1, 0])
>>> m = dft(8)
>>> m.dot(x)   # Compute the DFT of x
array([ 12.+0.j,  -2.-2.j,   0.-4.j,  -2.+2.j,   4.+0.j,  -2.-2.j,
        -0.+4.j,  -2.+2.j])
```

Verify that `m.dot(x)` is the same as `fft(x)`.

```
>>> from scipy.fftpack import fft
>>> fft(x)     # Same result as m.dot(x)
array([ 12.+0.j,  -2.-2.j,   0.-4.j,  -2.+2.j,   4.+0.j,  -2.-2.j,
         0.+4.j,  -2.+2.j])
```

`scipy.linalg.`**`hadamard`**`(`*n*, *dtype*=<*type 'int'*>`)`

Construct a Hadamard matrix.

Constructs an n-by-n Hadamard matrix, using Sylvester's construction. *n* must be a power of 2.

Parameters **n** : int
The order of the matrix. *n* must be a power of 2.
dtype : dtype, optional
The data type of the array to be constructed.

Returns **H** : (n, n) ndarray
The Hadamard matrix.

Notes

New in version 0.8.0.

Examples

```
>>> from scipy.linalg import hadamard
>>> hadamard(2, dtype=complex)
array([[ 1.+0.j,  1.+0.j],
       [ 1.+0.j, -1.-0.j]])
>>> hadamard(4)
array([[ 1,  1,  1,  1],
```

```
       [ 1, -1,  1, -1],
       [ 1,  1, -1, -1],
       [ 1, -1, -1,  1]])
```

`scipy.linalg.hankel(c, r=None)`

Construct a Hankel matrix.

The Hankel matrix has constant anti-diagonals, with *c* as its first column and *r* as its last row. If *r* is not given, then *r = zeros_like(c)* is assumed.

Parameters

c : array_like
First column of the matrix. Whatever the actual shape of *c*, it will be converted to a 1-D array.

r : array_like, optional
Last row of the matrix. If None, `r = zeros_like(c)` is assumed. r[0] is ignored; the last row of the returned matrix is `[c[-1], r[1:]]`. Whatever the actual shape of *r*, it will be converted to a 1-D array.

Returns

A : (len(c), len(r)) ndarray
The Hankel matrix. Dtype is the same as `(c[0] + r[0]).dtype`.

See also:

toeplitz Toeplitz matrix

circulant circulant matrix

Examples

```
>>> from scipy.linalg import hankel
>>> hankel([1, 17, 99])
array([[ 1, 17, 99],
       [17, 99,  0],
       [99,  0,  0]])
>>> hankel([1,2,3,4], [4,7,7,8,9])
array([[1, 2, 3, 4, 7],
       [2, 3, 4, 7, 7],
       [3, 4, 7, 7, 8],
       [4, 7, 7, 8, 9]])
```

`scipy.linalg.helmert(n, full=False)`

Create a Helmert matrix of order *n*.

This has applications in statistics, compositional or simplicial analysis, and in Aitchison geometry.

Parameters

n : int
The size of the array to create.

full : bool, optional
If True the (n, n) ndarray will be returned. Otherwise the submatrix that does not include the first row will be returned. Default: False.

Returns

M : ndarray
The Helmert matrix. The shape is (n, n) or (n-1, n) depending on the *full* argument.

Examples

```
>>> from scipy.linalg import helmert
>>> helmert(5, full=True)
array([[ 0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ,  0.4472136 ],
       [ 0.70710678, -0.70710678,  0.        ,  0.        ,  0.        ],
```

```
       [ 0.40824829,  0.40824829, -0.81649658,  0.        ,  0.        ],
       [ 0.28867513,  0.28867513,  0.28867513, -0.8660254 ,  0.        ],
       [ 0.2236068 ,  0.2236068 ,  0.2236068 ,  0.2236068 , -0.89442719]])
```

`scipy.linalg.`**`hilbert`**`(`*n*`)`

Create a Hilbert matrix of order *n*.

Returns the *n* by *n* array with entries *h[i,j] = 1 / (i + j + 1)*.

Parameters **n** : int

The size of the array to create.

Returns **h** : (n, n) ndarray

The Hilbert matrix.

See also:

`invhilbert` Compute the inverse of a Hilbert matrix.

Notes

New in version 0.10.0.

Examples

```
>>> from scipy.linalg import hilbert
>>> hilbert(3)
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.5       ,  0.33333333,  0.25      ],
       [ 0.33333333,  0.25      ,  0.2       ]])
```

`scipy.linalg.`**`invhilbert`**`(`*n*, *exact=False*`)`

Compute the inverse of the Hilbert matrix of order *n*.

The entries in the inverse of a Hilbert matrix are integers. When *n* is greater than 14, some entries in the inverse exceed the upper limit of 64 bit integers. The *exact* argument provides two options for dealing with these large integers.

Parameters **n** : int

The order of the Hilbert matrix.

exact : bool, optional

If False, the data type of the array that is returned is np.float64, and the array is an approximation of the inverse. If True, the array is the exact integer inverse array. To represent the exact inverse when n > 14, the returned array is an object array of long integers. For n <= 14, the exact inverse is returned as an array with data type np.int64.

Returns **invh** : (n, n) ndarray

The data type of the array is np.float64 if *exact* is False. If *exact* is True, the data type is either np.int64 (for n <= 14) or object (for n > 14). In the latter case, the objects in the array will be long integers.

See also:

`hilbert` Create a Hilbert matrix.

Notes

New in version 0.10.0.

Examples

```
>>> from scipy.linalg import invhilbert
>>> invhilbert(4)
array([[   16.,  -120.,   240.,  -140.],
       [ -120.,  1200., -2700.,  1680.],
       [  240., -2700.,  6480., -4200.],
       [ -140.,  1680., -4200.,  2800.]])
>>> invhilbert(4, exact=True)
array([[   16,  -120,   240,  -140],
       [ -120,  1200, -2700,  1680],
       [  240, -2700,  6480, -4200],
       [ -140,  1680, -4200,  2800]], dtype=int64)
>>> invhilbert(16)[7,7]
4.2475099528537506e+19
>>> invhilbert(16, exact=True)[7,7]
42475099528537378560L
```

`scipy.linalg.`**`leslie`**(*f*, *s*)

Create a Leslie matrix.

Given the length n array of fecundity coefficients *f* and the length n-1 array of survival coefficents `s`, return the associated Leslie matrix.

Parameters
- **f** : (N,) array_like
  The "fecundity" coefficients.
- **s** : (N-1,) array_like
  The "survival" coefficients, has to be 1-D. The length of `s` must be one less than the length of *f*, and it must be at least 1.

Returns
- **L** : (N, N) ndarray
  The array is zero except for the first row, which is *f*, and the first sub-diagonal, which is `s`. The data-type of the array will be the data-type of `f[0]+s[0]`.

Notes

New in version 0.8.0.

The Leslie matrix is used to model discrete-time, age-structured population growth *[R103] [R104]*. In a population with *n* age classes, two sets of parameters define a Leslie matrix: the *n* "fecundity coefficients", which give the number of offspring per-capita produced by each age class, and the *n* - 1 "survival coefficients", which give the per-capita survival rate of each age class.

References

[R103], *[R104]*

Examples

```
>>> from scipy.linalg import leslie
>>> leslie([0.1, 2.0, 1.0, 0.1], [0.2, 0.8, 0.7])
array([[ 0.1,  2. ,  1. ,  0.1],
       [ 0.2,  0. ,  0. ,  0. ],
       [ 0. ,  0.8,  0. ,  0. ],
       [ 0. ,  0. ,  0.7,  0. ]])
```

`scipy.linalg.`**`pascal`**(*n*, *kind='symmetric'*, *exact=True*)

Returns the n x n Pascal matrix.

The Pascal matrix is a matrix containing the binomial coefficients as its elements.

Parameters

n : int
The size of the matrix to create; that is, the result is an n x n matrix.

kind : str, optional
Must be one of 'symmetric', 'lower', or 'upper'. Default is 'symmetric'.

exact : bool, optional
If *exact* is True, the result is either an array of type numpy.uint64 (if n < 35) or an object array of Python long integers. If *exact* is False, the coefficients in the matrix are computed using `scipy.special.comb` with *exact=False*. The result will be a floating point array, and the values in the array will not be the exact coefficients, but this version is much faster than *exact=True*.

Returns

p : (n, n) ndarray
The Pascal matrix.

See also:

`invpascal`

Notes

See http://en.wikipedia.org/wiki/Pascal_matrix for more information about Pascal matrices.

New in version 0.11.0.

Examples

```
>>> from scipy.linalg import pascal
>>> pascal(4)
array([[ 1,  1,  1,  1],
       [ 1,  2,  3,  4],
       [ 1,  3,  6, 10],
       [ 1,  4, 10, 20]], dtype=uint64)
>>> pascal(4, kind='lower')
array([[1, 0, 0, 0],
       [1, 1, 0, 0],
       [1, 2, 1, 0],
       [1, 3, 3, 1]], dtype=uint64)
>>> pascal(50)[-1, -1]
25477612258980856902730428600L
>>> from scipy.special import comb
>>> comb(98, 49, exact=True)
25477612258980856902730428600L
```

`scipy.linalg.`**`invpascal`**(*n*, *kind='symmetric'*, *exact=True*)

Returns the inverse of the n x n Pascal matrix.

The Pascal matrix is a matrix containing the binomial coefficients as its elements.

Parameters

n : int
The size of the matrix to create; that is, the result is an n x n matrix.

kind : str, optional
Must be one of 'symmetric', 'lower', or 'upper'. Default is 'symmetric'.

exact : bool, optional
If *exact* is True, the result is either an array of type `numpy.int64` (if *n* <= 35) or an object array of Python integers. If *exact* is False, the coefficients in the matrix are computed using `scipy.special.comb` with *exact=False*. The result will be a floating point array, and for large *n*, the values in the array will not be the exact coefficients.

Returns

invp : (n, n) ndarray
The inverse of the Pascal matrix.

See also:

pascal

Notes

New in version 0.16.0.

References

[R101], [R102]

Examples

```
>>> from scipy.linalg import invpascal, pascal
>>> invp = invpascal(5)
>>> invp
array([[  5, -10,  10,  -5,   1],
       [-10,  30, -35,  19,  -4],
       [ 10, -35,  46, -27,   6],
       [ -5,  19, -27,  17,  -4],
       [  1,  -4,   6,  -4,   1]])
```

```
>>> p = pascal(5)
>>> p.dot(invp)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

An example of the use of *kind* and *exact*:

```
>>> invpascal(5, kind='lower', exact=False)
array([[ 1., -0.,  0., -0.,  0.],
       [-1.,  1., -0.,  0., -0.],
       [ 1., -2.,  1., -0.,  0.],
       [-1.,  3., -3.,  1., -0.],
       [ 1., -4.,  6., -4.,  1.]])
```

`scipy.linalg.`**`toeplitz`**`(`*c*, *r=None*`)`

Construct a Toeplitz matrix.

The Toeplitz matrix has constant diagonals, with c as its first column and r as its first row. If r is not given, `r == conjugate(c)` is assumed.

Parameters **c** : array_like

First column of the matrix. Whatever the actual shape of *c*, it will be converted to a 1-D array.

r : array_like, optional

First row of the matrix. If None, `r = conjugate(c)` is assumed; in this case, if c[0] is real, the result is a Hermitian matrix. r[0] is ignored; the first row of the returned matrix is `[c[0], r[1:]]`. Whatever the actual shape of *r*, it will be converted to a 1-D array.

Returns **A** : (len(c), len(r)) ndarray

The Toeplitz matrix. Dtype is the same as `(c[0] + r[0]).dtype`.

See also:

circulant circulant matrix

hankel Hankel matrix

Notes

The behavior when *c* or *r* is a scalar, or when *c* is complex and *r* is None, was changed in version 0.8.0. The behavior in previous versions was undocumented and is no longer supported.

Examples

```
>>> from scipy.linalg import toeplitz
>>> toeplitz([1,2,3], [1,4,5,6])
array([[1, 4, 5, 6],
       [2, 1, 4, 5],
       [3, 2, 1, 4]])
>>> toeplitz([1.0, 2+3j, 4-1j])
array([[ 1.+0.j,  2.-3.j,  4.+1.j],
       [ 2.+3.j,  1.+0.j,  2.-3.j],
       [ 4.-1.j,  2.+3.j,  1.+0.j]])
```

scipy.linalg.**tri**(*N*, *M=None*, *k=0*, *dtype=None*)

Construct (N, M) matrix filled with ones at and below the k-th diagonal.

The matrix has A[i,j] == 1 for i <= j + k

Parameters

N : int
: The size of the first dimension of the matrix.

M : int or None, optional
: The size of the second dimension of the matrix. If *M* is None, $M = N$ is assumed.

k : int, optional
: Number of subdiagonal below which matrix is filled with ones. $k = 0$ is the main diagonal, $k < 0$ subdiagonal and $k > 0$ superdiagonal.

dtype : dtype, optional
: Data type of the matrix.

Returns

tri : (N, M) ndarray
: Tri matrix.

Examples

```
>>> from scipy.linalg import tri
>>> tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1]])
>>> tri(3, 5, -1, dtype=int)
array([[0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [1, 1, 0, 0, 0]])
```

5.9.7 Low-level routines

get_blas_funcs(names[, arrays, dtype])	Return available BLAS function objects from names.
get_lapack_funcs(names[, arrays, dtype])	Return available LAPACK function objects from names.
find_best_blas_type([arrays, dtype])	Find best-matching BLAS/LAPACK type.

`scipy.linalg.`**`get_blas_funcs`**`(`*names*, *arrays=()*, *dtype=None*`)`

Return available BLAS function objects from names.

Arrays are used to determine the optimal prefix of BLAS routines.

Parameters **names** : str or sequence of str
Name(s) of BLAS functions without type prefix.
arrays : sequence of ndarrays, optional
Arrays can be given to determine optimal prefix of BLAS routines. If not given, double-precision routines will be used, otherwise the most generic type in arrays will be used.
dtype : str or dtype, optional
Data-type specifier. Not used if *arrays* is non-empty.

Returns **funcs** : list
List containing the found function(s).

Notes

This routine automatically chooses between Fortran/C interfaces. Fortran code is used whenever possible for arrays with column major order. In all other cases, C code is preferred.

In BLAS, the naming convention is that all functions start with a type prefix, which depends on the type of the principal matrix. These can be one of {'s', 'd', 'c', 'z'} for the numpy types {float32, float64, complex64, complex128} respectively. The code and the dtype are stored in attributes *typecode* and *dtype* of the returned functions.

`scipy.linalg.`**`get_lapack_funcs`**`(`*names*, *arrays=()*, *dtype=None*`)`

Return available LAPACK function objects from names.

Arrays are used to determine the optimal prefix of LAPACK routines.

Parameters **names** : str or sequence of str
Name(s) of LAPACK functions without type prefix.
arrays : sequence of ndarrays, optional
Arrays can be given to determine optimal prefix of LAPACK routines. If not given, double-precision routines will be used, otherwise the most generic type in arrays will be used.
dtype : str or dtype, optional
Data-type specifier. Not used if *arrays* is non-empty.

Returns **funcs** : list
List containing the found function(s).

Notes

This routine automatically chooses between Fortran/C interfaces. Fortran code is used whenever possible for arrays with column major order. In all other cases, C code is preferred.

In LAPACK, the naming convention is that all functions start with a type prefix, which depends on the type of the principal matrix. These can be one of {'s', 'd', 'c', 'z'} for the numpy types {float32, float64, complex64, complex128} respectevely, and are stored in attribute *typecode* of the returned functions.

`scipy.linalg.`**`find_best_blas_type`**`(`*arrays=()*, *dtype=None*`)`

Find best-matching BLAS/LAPACK type.

Arrays are used to determine the optimal prefix of BLAS routines.

Parameters **arrays** : sequence of ndarrays, optional
Arrays can be given to determine optimal prefix of BLAS routines. If not given, double-precision routines will be used, otherwise the most generic type in arrays will be used.
dtype : str or dtype, optional
Data-type specifier. Not used if *arrays* is non-empty.

Returns **prefix** : str
BLAS/LAPACK prefix character.
dtype : dtype
Inferred Numpy data type.
prefer_fortran : bool
Whether to prefer Fortran order routines over C order.

See also:

scipy.linalg.blas – Low-level BLAS functions

scipy.linalg.lapack – Low-level LAPACK functions

scipy.linalg.cython_blas – Low-level BLAS functions for Cython

scipy.linalg.cython_lapack – Low-level LAPACK functions for Cython

5.10 Low-level BLAS functions (scipy.linalg.blas)

This module contains low-level functions from the BLAS library.

New in version 0.12.0.

> **Warning:** These functions do little to no error checking. It is possible to cause crashes by mis-using them, so prefer using the higher-level routines in *scipy.linalg*.

5.10.1 Finding functions

get_blas_funcs(names[, arrays, dtype])	Return available BLAS function objects from names.
find_best_blas_type([arrays, dtype])	Find best-matching BLAS/LAPACK type.

scipy.linalg.blas.**get_blas_funcs**(*names*, *arrays=()*, *dtype=None*)
Return available BLAS function objects from names.

Arrays are used to determine the optimal prefix of BLAS routines.

Parameters **names** : str or sequence of str
Name(s) of BLAS functions without type prefix.
arrays : sequence of ndarrays, optional
Arrays can be given to determine optimal prefix of BLAS routines. If not given, double-precision routines will be used, otherwise the most generic type in arrays will be used.
dtype : str or dtype, optional
Data-type specifier. Not used if *arrays* is non-empty.

Returns **funcs** : list
List containing the found function(s).

Notes

This routine automatically chooses between Fortran/C interfaces. Fortran code is used whenever possible for arrays with column major order. In all other cases, C code is preferred.

In BLAS, the naming convention is that all functions start with a type prefix, which depends on the type of the principal matrix. These can be one of {‘s’, ‘d’, ‘c’, ‘z’} for the numpy types {float32, float64, complex64,

complex128} respectively. The code and the dtype are stored in attributes *typecode* and *dtype* of the returned functions.

scipy.linalg.blas.**find_best_blas_type**(*arrays=()*, *dtype=None*)

Find best-matching BLAS/LAPACK type.

Arrays are used to determine the optimal prefix of BLAS routines.

Parameters

arrays : sequence of ndarrays, optional
: Arrays can be given to determine optimal prefix of BLAS routines. If not given, double-precision routines will be used, otherwise the most generic type in arrays will be used.

dtype : str or dtype, optional
: Data-type specifier. Not used if *arrays* is non-empty.

Returns

prefix : str
: BLAS/LAPACK prefix character.

dtype : dtype
: Inferred Numpy data type.

prefer_fortran : bool
: Whether to prefer Fortran order routines over C order.

5.10.2 BLAS Level 1 functions

caxpy(x,y,[n,a,offx,incx,offy,incy])	Wrapper for caxpy.
ccopy(x,y,[n,offx,incx,offy,incy])	Wrapper for ccopy.
cdotc(x,y,[n,offx,incx,offy,incy])	Wrapper for cdotc.
cdotu(x,y,[n,offx,incx,offy,incy])	Wrapper for cdotu.
crotg(a,b)	Wrapper for crotg.
cscal(a,x,[n,offx,incx])	Wrapper for cscal.
csrot(...)	Wrapper for csrot.
csscal(a,x,[n,offx,incx,overwrite_x])	Wrapper for csscal.
cswap(x,y,[n,offx,incx,offy,incy])	Wrapper for cswap.
dasum(x,[n,offx,incx])	Wrapper for dasum.
daxpy(x,y,[n,a,offx,incx,offy,incy])	Wrapper for daxpy.
dcopy(x,y,[n,offx,incx,offy,incy])	Wrapper for dcopy.
ddot(x,y,[n,offx,incx,offy,incy])	Wrapper for ddot.
dnrm2(x,[n,offx,incx])	Wrapper for dnrm2.
drot(...)	Wrapper for drot.
drotg(a,b)	Wrapper for drotg.
drotm(...)	Wrapper for drotm.
drotmg(d1,d2,x1,y1)	Wrapper for drotmg.
dscal(a,x,[n,offx,incx])	Wrapper for dscal.
dswap(x,y,[n,offx,incx,offy,incy])	Wrapper for dswap.
dzasum(x,[n,offx,incx])	Wrapper for dzasum.
dznrm2(x,[n,offx,incx])	Wrapper for dznrm2.
icamax(x,[n,offx,incx])	Wrapper for icamax.
idamax(x,[n,offx,incx])	Wrapper for idamax.
isamax(x,[n,offx,incx])	Wrapper for isamax.
izamax(x,[n,offx,incx])	Wrapper for izamax.
sasum(x,[n,offx,incx])	Wrapper for sasum.
saxpy(x,y,[n,a,offx,incx,offy,incy])	Wrapper for saxpy.
scasum(x,[n,offx,incx])	Wrapper for scasum.
scnrm2(x,[n,offx,incx])	Wrapper for scnrm2.

Continued on next page

Table 5.81 – continued from previous page

scopy(x,y,[n,offx,incx,offy,incy])	Wrapper for scopy.
sdot(x,y,[n,offx,incx,offy,incy])	Wrapper for sdot.
snrm2(x,[n,offx,incx])	Wrapper for snrm2.
srot(...)	Wrapper for srot.
srotg(a,b)	Wrapper for srotg.
srotm(...)	Wrapper for srotm.
srotmg(d1,d2,x1,y1)	Wrapper for srotmg.
sscal(a,x,[n,offx,incx])	Wrapper for sscal.
sswap(x,y,[n,offx,incx,offy,incy])	Wrapper for sswap.
zaxpy(x,y,[n,a,offx,incx,offy,incy])	Wrapper for zaxpy.
zcopy(x,y,[n,offx,incx,offy,incy])	Wrapper for zcopy.
zdotc(x,y,[n,offx,incx,offy,incy])	Wrapper for zdotc.
zdotu(x,y,[n,offx,incx,offy,incy])	Wrapper for zdotu.
zdrot(...)	Wrapper for zdrot.
zdscal(a,x,[n,offx,incx,overwrite_x])	Wrapper for zdscal.
zrotg(a,b)	Wrapper for zrotg.
zscal(a,x,[n,offx,incx])	Wrapper for zscal.
zswap(x,y,[n,offx,incx,offy,incy])	Wrapper for zswap.

scipy.linalg.blas.**caxpy**(*x*, *y*[, *n*, *a*, *offx*, *incx*, *offy*, *incy*]) **= <fortran object>**

Wrapper for caxpy.

Parameters **x** : input rank-1 array('F') with bounds (*)

y : input rank-1 array('F') with bounds (*)

Returns **z** : rank-1 array('F') with bounds (*) and y storage

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

a : input complex, optional
Default: (1.0, 0.0)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

scipy.linalg.blas.**ccopy**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) **= <fortran object>**

Wrapper for ccopy.

Parameters **x** : input rank-1 array('F') with bounds (*)

y : input rank-1 array('F') with bounds (*)

Returns **y** : rank-1 array('F') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional

Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**cdotc**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = <fortran cdotc>
Wrapper for `cdotc`.

Parameters **x** : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)
Returns **xy** : complex
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**cdotu**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = <fortran cdotu>
Wrapper for `cdotu`.

Parameters **x** : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)
Returns **xy** : complex
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**crotg**(*a*, *b*) = <fortran object>
Wrapper for `crotg`.

Parameters **a** : input complex
b : input complex
Returns **c** : complex
s : complex

scipy.linalg.blas.**cscal**(*a*, *x*[, *n*, *offx*, *incx*]) = <fortran object>
Wrapper for `cscal`.

Parameters **a** : input complex
x : input rank-1 array('F') with bounds (*)
Returns **x** : rank-1 array('F') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional

Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**csrot** (*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) **= <fortran object>**

Wrapper for `csrot`.

Parameters **x** : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)
c : input float
s : input float

Returns **x** : rank-1 array('F') with bounds (*)
y : rank-1 array('F') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**csscal** (*a*, *x*[, *n*, *offx*, *incx*, *overwrite_x*]) **= <fortran object>**

Wrapper for `csscal`.

Parameters **a** : input float
x : input rank-1 array('F') with bounds (*)

Returns **x** : rank-1 array('F') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**cswap** (*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) **= <fortran object>**

Wrapper for `cswap`.

Parameters **x** : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)

Returns **x** : rank-1 array('F') with bounds (*)
y : rank-1 array('F') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional

Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**dasum**(*x*[, *n*, *offx*, *incx*]) = **<fortran dasum>**

Wrapper for dasum.

Parameters **x** : input rank-1 array('d') with bounds (*)

Returns **s** : float

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**daxpy**(*x*, *y*[, *n*, *a*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**

Wrapper for daxpy.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)

Returns **z** : rank-1 array('d') with bounds (*) and y storage

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)
a : input float, optional
Default: 1.0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**dcopy**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**

Wrapper for dcopy.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)

Returns **y** : rank-1 array('d') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional

Default: 1

`scipy.linalg.blas.`**`ddot`**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) **= <fortran ddot>**

Wrapper for `ddot`.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)

Returns **xy** : float

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`dnrm2`**(*x*[, *n*, *offx*, *incx*]) **= <fortran dnrm2>**

Wrapper for `dnrm2`.

Parameters **x** : input rank-1 array('d') with bounds (*)

Returns **n2** : float

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`drot`**(*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) **= <fortran object>**

Wrapper for `drot`.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)
c : input float
s : input float

Returns **x** : rank-1 array('d') with bounds (*)
y : rank-1 array('d') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0
offy : input int, optional
Default: 0
incy : input int, optional

Default: 1

scipy.linalg.blas.**drotg**(*a*, *b*) = <fortran object>

Wrapper for drotg.

Parameters **a** : input float
b : input float
Returns **c** : float
s : float

scipy.linalg.blas.**drotm**(*x*, *y*, *param*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) = <fortran object>

Wrapper for drotm.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)
param : input rank-1 array('d') with bounds (5)
Returns **x** : rank-1 array('d') with bounds (*)
y : rank-1 array('d') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**drotmg**(*d1*, *d2*, *x1*, *y1*) = <fortran object>

Wrapper for drotmg.

Parameters **d1** : input float
d2 : input float
x1 : input float
y1 : input float
Returns **param** : rank-1 array('d') with bounds (5)

scipy.linalg.blas.**dscal**(*a*, *x*[, *n*, *offx*, *incx*]) = <fortran object>

Wrapper for dscal.

Parameters **a** : input float
x : input rank-1 array('d') with bounds (*)
Returns **x** : rank-1 array('d') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**dswap**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = <fortran object>

Wrapper for `dswap`.

Parameters **x** : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)

Returns **x** : rank-1 array('d') with bounds (*)
y : rank-1 array('d') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**dzasum**(*x*[, *n*, *offx*, *incx*]) = <fortran dzasum>

Wrapper for `dzasum`.

Parameters **x** : input rank-1 array('D') with bounds (*)

Returns **s** : float

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**dznrm2**(*x*[, *n*, *offx*, *incx*]) = <fortran dznrm2>

Wrapper for `dznrm2`.

Parameters **x** : input rank-1 array('D') with bounds (*)

Returns **n2** : float

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.blas.**icamax**(*x*[, *n*, *offx*, *incx*]) = <fortran object>

Wrapper for `icamax`.

Parameters **x** : input rank-1 array('F') with bounds (*)

Returns **k** : int

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`idamax`**(*x*[, *n*, *offx*, *incx*]) = **<fortran object>**
Wrapper for `idamax`.

Parameters **x** : input rank-1 array('d') with bounds (*)
Returns **k** : int
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`isamax`**(*x*[, *n*, *offx*, *incx*]) = **<fortran object>**
Wrapper for `isamax`.

Parameters **x** : input rank-1 array('f') with bounds (*)
Returns **k** : int
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`izamax`**(*x*[, *n*, *offx*, *incx*]) = **<fortran object>**
Wrapper for `izamax`.

Parameters **x** : input rank-1 array('D') with bounds (*)
Returns **k** : int
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`sasum`**(*x*[, *n*, *offx*, *incx*]) = **<fortran sasum>**
Wrapper for `sasum`.

Parameters **x** : input rank-1 array('f') with bounds (*)
Returns **s** : float
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`saxpy`**(*x*, *y*[, *n*, *a*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**
Wrapper for `saxpy`.

Parameters **x** : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)
Returns **z** : rank-1 array('f') with bounds (*) and y storage

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

a : input float, optional
Default: 1.0

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`scasum`**(*x*[, *n*, *offx*, *incx*]) = **<fortran scasum>**

Wrapper for `scasum`.

Parameters **x** : input rank-1 array('F') with bounds (*)

Returns **s** : float

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`scnrm2`**(*x*[, *n*, *offx*, *incx*]) = **<fortran scnrm2>**

Wrapper for `scnrm2`.

Parameters **x** : input rank-1 array('F') with bounds (*)

Returns **n2** : float

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`scopy`**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**

Wrapper for `scopy`.

Parameters **x** : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)

Returns **y** : rank-1 array('f') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

scipy.linalg.blas.**sdot**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran sdot>**

Wrapper for `sdot`.

Parameters **x** : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)

Returns **xy** : float

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

scipy.linalg.blas.**snrm2**(*x*[, *n*, *offx*, *incx*]) = **<fortran snrm2>**

Wrapper for `snrm2`.

Parameters **x** : input rank-1 array('f') with bounds (*)

Returns **n2** : float

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

scipy.linalg.blas.**srot**(*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) = **<fortran object>**

Wrapper for `srot`.

Parameters **x** : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)
c : input float
s : input float

Returns **x** : rank-1 array('f') with bounds (*)
y : rank-1 array('f') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1

overwrite_x : input int, optional
Default: 0

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

overwrite_y : input int, optional
Default: 0

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`srotg`**(*a*, *b*) = **<fortran object>**

Wrapper for `srotg`.

Parameters **a** : input float
b : input float

Returns **c** : float
s : float

`scipy.linalg.blas.`**`srotm`**(*x*, *y*, *param*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) = **<fortran object>**

Wrapper for `srotm`.

Parameters **x** : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)
param : input rank-1 array('f') with bounds (5)

Returns **x** : rank-1 array('f') with bounds (*)
y : rank-1 array('f') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`srotmg`**(*d1*, *d2*, *x1*, *y1*) = **<fortran object>**

Wrapper for `srotmg`.

Parameters **d1** : input float
d2 : input float
x1 : input float
y1 : input float

Returns **param** : rank-1 array('f') with bounds (5)

`scipy.linalg.blas.`**`sscal`**(*a*, *x*[, *n*, *offx*, *incx*]) = **<fortran object>**

Wrapper for `sscal`.

Parameters **a** : input float
x : input rank-1 array('f') with bounds (*)

Returns **x** : rank-1 array('f') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`sswap`**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**

Wrapper for `sswap`.

Parameters
x : input rank-1 array('f') with bounds (*)
y : input rank-1 array('f') with bounds (*)

Returns
x : rank-1 array('f') with bounds (*)
y : rank-1 array('f') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**zaxpy**(*x*, *y*[, *n*, *a*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**
Wrapper for zaxpy.

Parameters
x : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)

Returns
z : rank-1 array('D') with bounds (*) and y storage

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
a : input complex, optional
Default: (1.0, 0.0)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**zcopy**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran object>**
Wrapper for zcopy.

Parameters
x : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)

Returns
y : rank-1 array('D') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

scipy.linalg.blas.**zdotc**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) = **<fortran zdotc>**
Wrapper for zdotc.

Parameters **x** : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)

Returns **xy** : complex

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`zdotu`**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) **= <fortran zdotu>**

Wrapper for `zdotu`.

Parameters **x** : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)

Returns **xy** : complex

Other Parameters

n : input int, optional
Default: (len(x)-offx)/abs(incx)

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`zdrot`**(*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) **= <fortran object>**

Wrapper for `zdrot`.

Parameters **x** : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)
c : input float
s : input float

Returns **x** : rank-1 array('D') with bounds (*)
y : rank-1 array('D') with bounds (*)

Other Parameters

n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1

overwrite_x : input int, optional
Default: 0

offx : input int, optional
Default: 0

incx : input int, optional
Default: 1

overwrite_y : input int, optional
Default: 0

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

`scipy.linalg.blas.`**`zdscal`**(*a*, *x*[, *n*, *offx*, *incx*, *overwrite_x*]) **= <fortran object>**
Wrapper for `zdscal`.

Parameters **a** : input float
x : input rank-1 array('D') with bounds (*)
Returns **x** : rank-1 array('D') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`zrotg`**(*a*, *b*) **= <fortran object>**
Wrapper for `zrotg`.

Parameters **a** : input complex
b : input complex
Returns **c** : complex
s : complex

`scipy.linalg.blas.`**`zscal`**(*a*, *x*[, *n*, *offx*, *incx*]) **= <fortran object>**
Wrapper for `zscal`.

Parameters **a** : input complex
x : input rank-1 array('D') with bounds (*)
Returns **x** : rank-1 array('D') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.blas.`**`zswap`**(*x*, *y*[, *n*, *offx*, *incx*, *offy*, *incy*]) **= <fortran object>**
Wrapper for `zswap`.

Parameters **x** : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)
Returns **x** : rank-1 array('D') with bounds (*)
y : rank-1 array('D') with bounds (*)
Other Parameters
n : input int, optional
Default: (len(x)-offx)/abs(incx)
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional

Default: 1

5.10.3 BLAS Level 2 functions

cgemv(...)	Wrapper for cgemv.
cgerc(...)	Wrapper for cgerc.
cgeru(...)	Wrapper for cgeru.
chemv(...)	Wrapper for chemv.
ctrmv(...)	Wrapper for ctrmv.
csyr(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for csyr.
cher(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for cher.
cher2(...)	Wrapper for cher2.
dgemv(...)	Wrapper for dgemv.
dger(...)	Wrapper for dger.
dsymv(...)	Wrapper for dsymv.
dtrmv(...)	Wrapper for dtrmv.
dsyr(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for dsyr.
dsyr2(...)	Wrapper for dsyr2.
sgemv(...)	Wrapper for sgemv.
sger(...)	Wrapper for sger.
ssymv(...)	Wrapper for ssymv.
strmv(...)	Wrapper for strmv.
ssyr(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for ssyr.
ssyr2(...)	Wrapper for ssyr2.
zgemv(...)	Wrapper for zgemv.
zgerc(...)	Wrapper for zgerc.
zgeru(...)	Wrapper for zgeru.
zhemv(...)	Wrapper for zhemv.
ztrmv(...)	Wrapper for ztrmv.
zsyr(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for zsyr.
zher(alpha,x,[lower,incx,offx,n,a,overwrite_a])	Wrapper for zher.
zher2(...)	Wrapper for zher2.

scipy.linalg.blas.**cgemv**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *trans*, *overwrite_y*]) **= <fortran object>**

Wrapper for cgemv.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (m,n)
x : input rank-1 array('F') with bounds (*)

Returns **y** : rank-1 array('F') with bounds (ly)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
y : input rank-1 array('F') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1

offy : input int, optional
    Default: 0
incy : input int, optional
    Default: 1
trans : input int, optional
    Default: 0

scipy.linalg.blas.**cgerc**(*alpha*, *x*, *y*[, *incx*, *incy*, *a*, *overwrite_x*, *overwrite_y*, *overwrite_a*]) = <fortran object>

Wrapper for `cgerc`.

Parameters **alpha** : input complex
x : input rank-1 array('F') with bounds (m)
y : input rank-1 array('F') with bounds (n)

Returns **a** : rank-2 array('F') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
    Default: 1
incx : input int, optional
    Default: 1
overwrite_y : input int, optional
    Default: 1
incy : input int, optional
    Default: 1
a : input rank-2 array('F') with bounds (m,n), optional
    Default: (0.0,0.0)
overwrite_a : input int, optional
    Default: 0

scipy.linalg.blas.**cgeru**(*alpha*, *x*, *y*[, *incx*, *incy*, *a*, *overwrite_x*, *overwrite_y*, *overwrite_a*]) = <fortran object>

Wrapper for `cgeru`.

Parameters **alpha** : input complex
x : input rank-1 array('F') with bounds (m)
y : input rank-1 array('F') with bounds (n)

Returns **a** : rank-2 array('F') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
    Default: 1
incx : input int, optional
    Default: 1
overwrite_y : input int, optional
    Default: 1
incy : input int, optional
    Default: 1
a : input rank-2 array('F') with bounds (m,n), optional
    Default: (0.0,0.0)
overwrite_a : input int, optional
    Default: 0

scipy.linalg.blas.**chemv**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *lower*, *overwrite_y*]) = <fortran object>

Wrapper for `chemv`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (n,n)

x : input rank-1 array('F') with bounds (*)

Returns **y** : rank-1 array('F') with bounds (ly)

Other Parameters

beta : input complex, optional
Default: (0.0, 0.0)
y : input rank-1 array('F') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1
lower : input int, optional
Default: 0

scipy.linalg.blas.**ctrmv**(*a*, *x*[, *offx*, *incx*, *lower*, *trans*, *unitdiag*, *overwrite_x*]) **= <fortran object>**

Wrapper for `ctrmv`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
x : input rank-1 array('F') with bounds (*)

Returns **x** : rank-1 array('F') with bounds (*)

Other Parameters

overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
lower : input int, optional
Default: 0
trans : input int, optional
Default: 0
unitdiag : input int, optional
Default: 0

scipy.linalg.blas.**csyr**(*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) **= <fortran object>**

Wrapper for `csyr`.

Parameters **alpha** : input complex
x : input rank-1 array('F') with bounds (*)

Returns **a** : rank-2 array('F') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
a : input rank-2 array('F') with bounds (n,n)
overwrite_a : input int, optional

Default: 0

scipy.linalg.blas.**cher** (*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cher`.

Parameters **alpha** : input complex
x : input rank-1 array('F') with bounds (*)

Returns **a** : rank-2 array('F') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
a : input rank-2 array('F') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**cher2** (*alpha*, *x*, *y*[, *lower*, *incx*, *offx*, *incy*, *offy*, *n*, *a*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cher2`.

Parameters **alpha** : input complex
x : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)

Returns **a** : rank-2 array('F') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
incy : input int, optional
Default: 1
offy : input int, optional
Default: 0
n : input int, optional
Default: ((len(x)-1-offx)/abs(incx)+1 <=(len(y)-1-offy)/abs(incy)+1 ?(len(x)-1-offx)/abs(incx)+1 :(len(y)-1-offy)/abs(incy)+1)
a : input rank-2 array('F') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**dgemv** (*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *trans*, *overwrite_y*]) **= <fortran object>**

Wrapper for `dgemv`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (m,n)
x : input rank-1 array('d') with bounds (*)

Returns **y** : rank-1 array('d') with bounds (ly)

Other Parameters

beta : input float, optional

Default: 0.0
y : input rank-1 array('d') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1
trans : input int, optional
Default: 0

scipy.linalg.blas.**dger**(*alpha, x, y*[, *incx, incy, a, overwrite_x, overwrite_y, overwrite_a*]) = <fortran object>

Wrapper for `dger`.

Parameters **alpha** : input float
x : input rank-1 array('d') with bounds (m)
y : input rank-1 array('d') with bounds (n)

Returns **a** : rank-2 array('d') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
Default: 1
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 1
incy : input int, optional
Default: 1
a : input rank-2 array('d') with bounds (m,n), optional
Default: 0.0
overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**dsymv**(*alpha, a, x*[, *beta, y, offx, incx, offy, incy, lower, overwrite_y*]) = <fortran object>

Wrapper for `dsymv`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (n,n)
x : input rank-1 array('d') with bounds (*)

Returns **y** : rank-1 array('d') with bounds (ly)

Other Parameters
beta : input float, optional
Default: 0.0
y : input rank-1 array('d') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional

Default: 0
incy : input int, optional
Default: 1
lower : input int, optional
Default: 0

scipy.linalg.blas.**dtrmv**(*a*, *x*[, *offx*, *incx*, *lower*, *trans*, *unitdiag*, *overwrite_x*]) = <fortran object>
Wrapper for dtrmv.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
x : input rank-1 array('d') with bounds (*)
Returns **x** : rank-1 array('d') with bounds (*)
Other Parameters
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
lower : input int, optional
Default: 0
trans : input int, optional
Default: 0
unitdiag : input int, optional
Default: 0

scipy.linalg.blas.**dsyr**(*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) = <fortran object>
Wrapper for dsyr.

Parameters **alpha** : input float
x : input rank-1 array('d') with bounds (*)
Returns **a** : rank-2 array('d') with bounds (n,n)
Other Parameters
lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
a : input rank-2 array('d') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**dsyr2**(*alpha*, *x*, *y*[, *lower*, *incx*, *offx*, *incy*, *offy*, *n*, *a*, *overwrite_a*]) = <fortran object>
Wrapper for dsyr2.

Parameters **alpha** : input float
x : input rank-1 array('d') with bounds (*)
y : input rank-1 array('d') with bounds (*)
Returns **a** : rank-2 array('d') with bounds (n,n)
Other Parameters
lower : input int, optional
Default: 0
incx : input int, optional

Default: 1
offx : input int, optional
Default: 0
incy : input int, optional
Default: 1
offy : input int, optional
Default: 0
n : input int, optional
Default: ((len(x)-1-offx)/abs(incx)+1 <=(len(y)-1-offy)/abs(incy)+1 ?(len(x)-1-offx)/abs(incx)+1 :(len(y)-1-offy)/abs(incy)+1)
a : input rank-2 array('d') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

`scipy.linalg.blas.`**`sgemv`**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *trans*, *overwrite_y*]) **= <fortran object>**

Wrapper for `sgemv`.

Parameters **alpha** : input float
a : input rank-2 array('f') with bounds (m,n)
x : input rank-1 array('f') with bounds (*)

Returns **y** : rank-1 array('f') with bounds (ly)

Other Parameters
beta : input float, optional
Default: 0.0
y : input rank-1 array('f') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1
trans : input int, optional
Default: 0

`scipy.linalg.blas.`**`sger`**(*alpha*, *x*, *y*[, *incx*, *incy*, *a*, *overwrite_x*, *overwrite_y*, *overwrite_a*]) **= <fortran object>**

Wrapper for `sger`.

Parameters **alpha** : input float
x : input rank-1 array('f') with bounds (m)
y : input rank-1 array('f') with bounds (n)

Returns **a** : rank-2 array('f') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
Default: 1
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 1
incy : input int, optional
Default: 1

a : input rank-2 array('f') with bounds (m,n), optional
Default: 0.0
overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**ssymv**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *lower*, *overwrite_y*]) = <fortran object>

Wrapper for ssymv.

Parameters **alpha** : input float
a : input rank-2 array('f') with bounds (n,n)
x : input rank-1 array('f') with bounds (*)

Returns **y** : rank-1 array('f') with bounds (ly)

Other Parameters
beta : input float, optional
Default: 0.0
y : input rank-1 array('f') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1
lower : input int, optional
Default: 0

scipy.linalg.blas.**strmv**(*a*, *x*[, *offx*, *incx*, *lower*, *trans*, *unitdiag*, *overwrite_x*]) = <fortran object>

Wrapper for strmv.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
x : input rank-1 array('f') with bounds (*)

Returns **x** : rank-1 array('f') with bounds (*)

Other Parameters
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
lower : input int, optional
Default: 0
trans : input int, optional
Default: 0
unitdiag : input int, optional
Default: 0

scipy.linalg.blas.**ssyr**(*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) = <fortran object>

Wrapper for ssyr.

Parameters **alpha** : input float
x : input rank-1 array('f') with bounds (*)

Returns **a** : rank-2 array('f') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0

incx : input int, optional
Default: 1

offx : input int, optional
Default: 0

n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1

a : input rank-2 array('f') with bounds (n,n)

overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**ssyr2**(*alpha*, *x*, *y*[, *lower*, *incx*, *offx*, *incy*, *offy*, *n*, *a*, *overwrite_a*]) = **<fortran object>**

Wrapper for ssyr2.

Parameters

alpha : input float

x : input rank-1 array('f') with bounds (*)

y : input rank-1 array('f') with bounds (*)

Returns

a : rank-2 array('f') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0

incx : input int, optional
Default: 1

offx : input int, optional
Default: 0

incy : input int, optional
Default: 1

offy : input int, optional
Default: 0

n : input int, optional
Default: ((len(x)-1-offx)/abs(incx)+1 <=(len(y)-1-offy)/abs(incy)+1 ?(len(x)-1-offx)/abs(incx)+1 :(len(y)-1-offy)/abs(incy)+1)

a : input rank-2 array('f') with bounds (n,n)

overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**zgemv**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *trans*, *overwrite_y*]) = **<fortran object>**

Wrapper for zgemv.

Parameters

alpha : input complex

a : input rank-2 array('D') with bounds (m,n)

x : input rank-1 array('D') with bounds (*)

Returns

y : rank-1 array('D') with bounds (ly)

Other Parameters

beta : input complex, optional
Default: (0.0, 0.0)

y : input rank-1 array('D') with bounds (ly)

overwrite_y : input int, optional
Default: 0

offx : input int, optional
Default: 0

incx : input int, optional

Default: 1

offy : input int, optional
Default: 0

incy : input int, optional
Default: 1

trans : input int, optional
Default: 0

scipy.linalg.blas.**zgerc**(*alpha*, *x*, *y*[, *incx*, *incy*, *a*, *overwrite_x*, *overwrite_y*, *overwrite_a*]) = <fortran object>

Wrapper for zgerc.

Parameters **alpha** : input complex
x : input rank-1 array('D') with bounds (m)
y : input rank-1 array('D') with bounds (n)

Returns **a** : rank-2 array('D') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
Default: 1

incx : input int, optional
Default: 1

overwrite_y : input int, optional
Default: 1

incy : input int, optional
Default: 1

a : input rank-2 array('D') with bounds (m,n), optional
Default: (0.0,0.0)

overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**zgeru**(*alpha*, *x*, *y*[, *incx*, *incy*, *a*, *overwrite_x*, *overwrite_y*, *overwrite_a*]) = <fortran object>

Wrapper for zgeru.

Parameters **alpha** : input complex
x : input rank-1 array('D') with bounds (m)
y : input rank-1 array('D') with bounds (n)

Returns **a** : rank-2 array('D') with bounds (m,n)

Other Parameters
overwrite_x : input int, optional
Default: 1

incx : input int, optional
Default: 1

overwrite_y : input int, optional
Default: 1

incy : input int, optional
Default: 1

a : input rank-2 array('D') with bounds (m,n), optional
Default: (0.0,0.0)

overwrite_a : input int, optional
Default: 0

scipy.linalg.blas.**zhemv**(*alpha*, *a*, *x*[, *beta*, *y*, *offx*, *incx*, *offy*, *incy*, *lower*, *overwrite_y*]) = <fortran object>

Wrapper for zhemv.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (n,n)
x : input rank-1 array('D') with bounds (*)
Returns **y** : rank-1 array('D') with bounds (ly)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
y : input rank-1 array('D') with bounds (ly)
overwrite_y : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1
lower : input int, optional
Default: 0

`scipy.linalg.blas.`**`ztrmv`**(*a*, *x*[, *offx*, *incx*, *lower*, *trans*, *unitdiag*, *overwrite_x*]) **= <fortran object>**
Wrapper for `ztrmv`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
x : input rank-1 array('D') with bounds (*)
Returns **x** : rank-1 array('D') with bounds (*)
Other Parameters
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
lower : input int, optional
Default: 0
trans : input int, optional
Default: 0
unitdiag : input int, optional
Default: 0

`scipy.linalg.blas.`**`zsyr`**(*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zsyr`.

Parameters **alpha** : input complex
x : input rank-1 array('D') with bounds (*)
Returns **a** : rank-2 array('D') with bounds (n,n)
Other Parameters
lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1

a : input rank-2 array('D') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

`scipy.linalg.blas.`**`zher`**(*alpha*, *x*[, *lower*, *incx*, *offx*, *n*, *a*, *overwrite_a*]) **= <fortran object>**

Wrapper for `zher`.

Parameters **alpha** : input complex
x : input rank-1 array('D') with bounds (*)

Returns **a** : rank-2 array('D') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
a : input rank-2 array('D') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

`scipy.linalg.blas.`**`zher2`**(*alpha*, *x*, *y*[, *lower*, *incx*, *offx*, *incy*, *offy*, *n*, *a*, *overwrite_a*]) **= <fortran object>**

Wrapper for `zher2`.

Parameters **alpha** : input complex
x : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)

Returns **a** : rank-2 array('D') with bounds (n,n)

Other Parameters

lower : input int, optional
Default: 0
incx : input int, optional
Default: 1
offx : input int, optional
Default: 0
incy : input int, optional
Default: 1
offy : input int, optional
Default: 0
n : input int, optional
Default: ((len(x)-1-offx)/abs(incx)+1 <=(len(y)-1-offy)/abs(incy)+1 ?(len(x)-1-offx)/abs(incx)+1 :(len(y)-1-offy)/abs(incy)+1)
a : input rank-2 array('D') with bounds (n,n)
overwrite_a : input int, optional
Default: 0

5.10.4 BLAS Level 3 functions

cgemm(...)	Wrapper for `cgemm`.
chemm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for `chemm`.

Continued on next page

Table 5.83 – continued from previous page

cherk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for cherk.
cher2k(...)	Wrapper for cher2k.
csymm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for csymm.
csyrk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for csyrk.
csyr2k(...)	Wrapper for csyr2k.
dgemm(...)	Wrapper for dgemm.
dsymm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for dsymm.
dsyrk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for dsyrk.
dsyr2k(...)	Wrapper for dsyr2k.
sgemm(...)	Wrapper for sgemm.
ssymm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for ssymm.
ssyrk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for ssyrk.
ssyr2k(...)	Wrapper for ssyr2k.
zgemm(...)	Wrapper for zgemm.
zhemm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for zhemm.
zherk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for zherk.
zher2k(...)	Wrapper for zher2k.
zsymm(alpha,a,b,[beta,c,side,lower,overwrite_c])	Wrapper for zsymm.
zsyrk(alpha,a,[beta,c,trans,lower,overwrite_c])	Wrapper for zsyrk.
zsyr2k(...)	Wrapper for zsyr2k.

scipy.linalg.blas.**cgemm**(*alpha*, *a*, *b*[, *beta*, *c*, *trans_a*, *trans_b*, *overwrite_c*]) **= <fortran object>**

Wrapper for cgemm.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
b : input rank-2 array('F') with bounds (ldb,kb)

Returns **c** : rank-2 array('F') with bounds (m,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
trans_a : input int, optional
Default: 0
trans_b : input int, optional
Default: 0

scipy.linalg.blas.**chemm**(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) **= <fortran object>**

Wrapper for chemm.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
b : input rank-2 array('F') with bounds (ldb,kb)

Returns **c** : rank-2 array('F') with bounds (m,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
side : input int, optional

Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.cherk`(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = **<fortran object>**
Wrapper for `cherk`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
Returns **c** : rank-2 array('F') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.cher2k`(*alpha*, *a*, *b*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = **<fortran object>**
Wrapper for `cher2k`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
b : input rank-2 array('F') with bounds (ldb,kb)
Returns **c** : rank-2 array('F') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.csymm`(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) = **<fortran object>**
Wrapper for `csymm`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
b : input rank-2 array('F') with bounds (ldb,kb)
Returns **c** : rank-2 array('F') with bounds (m,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
side : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**csyrk**(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for `csyrk`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
Returns **c** : rank-2 array('F') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**csyr2k**(*alpha*, *a*, *b*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for `csyr2k`.

Parameters **alpha** : input complex
a : input rank-2 array('F') with bounds (lda,ka)
b : input rank-2 array('F') with bounds (ldb,kb)
Returns **c** : rank-2 array('F') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('F') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**dgemm**(*alpha*, *a*, *b*[, *beta*, *c*, *trans_a*, *trans_b*, *overwrite_c*]) = <fortran object>
Wrapper for `dgemm`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (lda,ka)
b : input rank-2 array('d') with bounds (ldb,kb)
Returns **c** : rank-2 array('d') with bounds (m,n)
Other Parameters
beta : input float, optional
Default: 0.0
c : input rank-2 array('d') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
trans_a : input int, optional
Default: 0
trans_b : input int, optional
Default: 0

scipy.linalg.blas.**dsymm**(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for `dsymm`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (lda,ka)
b : input rank-2 array('d') with bounds (ldb,kb)
Returns **c** : rank-2 array('d') with bounds (m,n)
Other Parameters
beta : input float, optional
Default: 0.0
c : input rank-2 array('d') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
side : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.`**`dsyrk`**(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) **= <fortran object>**
Wrapper for `dsyrk`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (lda,ka)
Returns **c** : rank-2 array('d') with bounds (n,n)
Other Parameters
beta : input float, optional
Default: 0.0
c : input rank-2 array('d') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.`**`dsyr2k`**(*alpha*, *a*, *b*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) **= <fortran object>**
Wrapper for `dsyr2k`.

Parameters **alpha** : input float
a : input rank-2 array('d') with bounds (lda,ka)
b : input rank-2 array('d') with bounds (ldb,kb)
Returns **c** : rank-2 array('d') with bounds (n,n)
Other Parameters
beta : input float, optional
Default: 0.0
c : input rank-2 array('d') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.blas.`**`sgemm`**(*alpha*, *a*, *b*[, *beta*, *c*, *trans_a*, *trans_b*, *overwrite_c*]) **= <fortran object>**
Wrapper for `sgemm`.

Parameters **alpha** : input float
a : input rank-2 array('f') with bounds (lda,ka)
b : input rank-2 array('f') with bounds (ldb,kb)
Returns **c** : rank-2 array('f') with bounds (m,n)

Other Parameters

beta : input float, optional
Default: 0.0

c : input rank-2 array('f') with bounds (m,n)

overwrite_c : input int, optional
Default: 0

trans_a : input int, optional
Default: 0

trans_b : input int, optional
Default: 0

scipy.linalg.blas.**ssymm**(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) = **<fortran object>**

Wrapper for `ssymm`.

Parameters

alpha : input float

a : input rank-2 array('f') with bounds (lda,ka)

b : input rank-2 array('f') with bounds (ldb,kb)

Returns

c : rank-2 array('f') with bounds (m,n)

Other Parameters

beta : input float, optional
Default: 0.0

c : input rank-2 array('f') with bounds (m,n)

overwrite_c : input int, optional
Default: 0

side : input int, optional
Default: 0

lower : input int, optional
Default: 0

scipy.linalg.blas.**ssyrk**(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = **<fortran object>**

Wrapper for `ssyrk`.

Parameters

alpha : input float

a : input rank-2 array('f') with bounds (lda,ka)

Returns

c : rank-2 array('f') with bounds (n,n)

Other Parameters

beta : input float, optional
Default: 0.0

c : input rank-2 array('f') with bounds (n,n)

overwrite_c : input int, optional
Default: 0

trans : input int, optional
Default: 0

lower : input int, optional
Default: 0

scipy.linalg.blas.**ssyr2k**(*alpha*, *a*, *b*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = **<fortran object>**

Wrapper for `ssyr2k`.

Parameters

alpha : input float

a : input rank-2 array('f') with bounds (lda,ka)

b : input rank-2 array('f') with bounds (ldb,kb)

Returns

c : rank-2 array('f') with bounds (n,n)

Other Parameters

beta : input float, optional
Default: 0.0

c : input rank-2 array('f') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zgemm**(*alpha*, *a*, *b*[, *beta*, *c*, *trans_a*, *trans_b*, *overwrite_c*]) = <fortran object>
Wrapper for zgemm.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
b : input rank-2 array('D') with bounds (ldb,kb)

Returns **c** : rank-2 array('D') with bounds (m,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
trans_a : input int, optional
Default: 0
trans_b : input int, optional
Default: 0

scipy.linalg.blas.**zhemm**(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zhemm.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
b : input rank-2 array('D') with bounds (ldb,kb)

Returns **c** : rank-2 array('D') with bounds (m,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
side : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zherk**(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zherk.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)

Returns **c** : rank-2 array('D') with bounds (n,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional

Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zher2k**(*alpha*, *a*, *b*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zher2k.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
b : input rank-2 array('D') with bounds (ldb,kb)
Returns **c** : rank-2 array('D') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zsymm**(*alpha*, *a*, *b*[, *beta*, *c*, *side*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zsymm.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
b : input rank-2 array('D') with bounds (ldb,kb)
Returns **c** : rank-2 array('D') with bounds (m,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (m,n)
overwrite_c : input int, optional
Default: 0
side : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zsyrk**(*alpha*, *a*[, *beta*, *c*, *trans*, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zsyrk.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
Returns **c** : rank-2 array('D') with bounds (n,n)
Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.blas.**zsyr2k**(*alpha, a, b*[, *beta, c, trans, lower, overwrite_c*]) = <fortran object>
Wrapper for zsyr2k.

Parameters **alpha** : input complex
a : input rank-2 array('D') with bounds (lda,ka)
b : input rank-2 array('D') with bounds (ldb,kb)

Returns **c** : rank-2 array('D') with bounds (n,n)

Other Parameters
beta : input complex, optional
Default: (0.0, 0.0)
c : input rank-2 array('D') with bounds (n,n)
overwrite_c : input int, optional
Default: 0
trans : input int, optional
Default: 0
lower : input int, optional
Default: 0

5.11 Low-level LAPACK functions (scipy.linalg.lapack)

This module contains low-level functions from the LAPACK library.

The **gegv* family of routines have been removed from LAPACK 3.6.0 and have been deprecated in SciPy 0.17.0. They will be removed in a future release.

New in version 0.12.0.

Warning: These functions do little to no error checking. It is possible to cause crashes by mis-using them, so prefer using the higher-level routines in *scipy.linalg*.

5.11.1 Finding functions

get_lapack_funcs(names[, arrays, dtype])	Return available LAPACK function objects from names.

5.11.2 All functions

sgbsv(kl,ku,ab,b,[overwrite_ab,overwrite_b])	Wrapper for sgbsv.
dgbsv(kl,ku,ab,b,[overwrite_ab,overwrite_b])	Wrapper for dgbsv.
cgbsv(kl,ku,ab,b,[overwrite_ab,overwrite_b])	Wrapper for cgbsv.
zgbsv(kl,ku,ab,b,[overwrite_ab,overwrite_b])	Wrapper for zgbsv.
sgbtrf(ab,kl,ku,[m,n,ldab,overwrite_ab])	Wrapper for sgbtrf.
dgbtrf(ab,kl,ku,[m,n,ldab,overwrite_ab])	Wrapper for dgbtrf.
cgbtrf(ab,kl,ku,[m,n,ldab,overwrite_ab])	Wrapper for cgbtrf.
zgbtrf(ab,kl,ku,[m,n,ldab,overwrite_ab])	Wrapper for zgbtrf.
sgbtrs(...)	Wrapper for sgbtrs.
dgbtrs(...)	Wrapper for dgbtrs.
cgbtrs(...)	Wrapper for cgbtrs.
zgbtrs(...)	Wrapper for zgbtrs.

Continued on next page

Table 5.85 – continued from previous page

sgebal(a,[scale,permute,overwrite_a])	Wrapper for `sgebal`.
dgebal(a,[scale,permute,overwrite_a])	Wrapper for `dgebal`.
cgebal(a,[scale,permute,overwrite_a])	Wrapper for `cgebal`.
zgebal(a,[scale,permute,overwrite_a])	Wrapper for `zgebal`.
sgees(...)	Wrapper for `sgees`.
dgees(...)	Wrapper for `dgees`.
cgees(...)	Wrapper for `cgees`.
zgees(...)	Wrapper for `zgees`.
sgeev(...)	Wrapper for `sgeev`.
dgeev(...)	Wrapper for `dgeev`.
cgeev(...)	Wrapper for `cgeev`.
zgeev(...)	Wrapper for `zgeev`.
sgeev_lwork(n,[compute_vl,compute_vr])	Wrapper for `sgeev_lwork`.
dgeev_lwork(n,[compute_vl,compute_vr])	Wrapper for `dgeev_lwork`.
cgeev_lwork(n,[compute_vl,compute_vr])	Wrapper for `cgeev_lwork`.
zgeev_lwork(n,[compute_vl,compute_vr])	Wrapper for `zgeev_lwork`.
sgegv(*args, **kwds)	*sgegv* is deprecated!
dgegv(*args, **kwds)	*dgegv* is deprecated!
cgegv(*args, **kwds)	*cgegv* is deprecated!
zgegv(*args, **kwds)	*zgegv* is deprecated!
sgehrd(a,[lo,hi,lwork,overwrite_a])	Wrapper for `sgehrd`.
dgehrd(a,[lo,hi,lwork,overwrite_a])	Wrapper for `dgehrd`.
cgehrd(a,[lo,hi,lwork,overwrite_a])	Wrapper for `cgehrd`.
zgehrd(a,[lo,hi,lwork,overwrite_a])	Wrapper for `zgehrd`.
sgehrd_lwork(n,[lo,hi])	Wrapper for `sgehrd_lwork`.
dgehrd_lwork(n,[lo,hi])	Wrapper for `dgehrd_lwork`.
cgehrd_lwork(n,[lo,hi])	Wrapper for `cgehrd_lwork`.
zgehrd_lwork(n,[lo,hi])	Wrapper for `zgehrd_lwork`.
sgelss(a,b,[cond,lwork,overwrite_a,overwrite_b])	Wrapper for `sgelss`.
dgelss(a,b,[cond,lwork,overwrite_a,overwrite_b])	Wrapper for `dgelss`.
cgelss(a,b,[cond,lwork,overwrite_a,overwrite_b])	Wrapper for `cgelss`.
zgelss(a,b,[cond,lwork,overwrite_a,overwrite_b])	Wrapper for `zgelss`.
sgelss_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `sgelss_lwork`.
dgelss_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `dgelss_lwork`.
cgelss_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `cgelss_lwork`.
zgelss_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `zgelss_lwork`.
sgelsd(...)	Wrapper for `sgelsd`.
dgelsd(...)	Wrapper for `dgelsd`.
cgelsd(...)	Wrapper for `cgelsd`.
zgelsd(...)	Wrapper for `zgelsd`.
sgelsd_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `sgelsd_lwork`.
dgelsd_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `dgelsd_lwork`.
cgelsd_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `cgelsd_lwork`.
zgelsd_lwork(m,n,nrhs,[cond,lwork])	Wrapper for `zgelsd_lwork`.
sgelsy(...)	Wrapper for `sgelsy`.
dgelsy(...)	Wrapper for `dgelsy`.
cgelsy(...)	Wrapper for `cgelsy`.
zgelsy(...)	Wrapper for `zgelsy`.
sgelsy_lwork(m,n,nrhs,cond,[lwork])	Wrapper for `sgelsy_lwork`.
dgelsy_lwork(m,n,nrhs,cond,[lwork])	Wrapper for `dgelsy_lwork`.

Continued on next page

Table 5.85 – continued from previous page

cgelsy_lwork(m,n,nrhs,cond,[lwork])	Wrapper for cgelsy_lwork.
zgelsy_lwork(m,n,nrhs,cond,[lwork])	Wrapper for zgelsy_lwork.
sgeqp3(a,[lwork,overwrite_a])	Wrapper for sgeqp3.
dgeqp3(a,[lwork,overwrite_a])	Wrapper for dgeqp3.
cgeqp3(a,[lwork,overwrite_a])	Wrapper for cgeqp3.
zgeqp3(a,[lwork,overwrite_a])	Wrapper for zgeqp3.
sgeqrf(a,[lwork,overwrite_a])	Wrapper for sgeqrf.
dgeqrf(a,[lwork,overwrite_a])	Wrapper for dgeqrf.
cgeqrf(a,[lwork,overwrite_a])	Wrapper for cgeqrf.
zgeqrf(a,[lwork,overwrite_a])	Wrapper for zgeqrf.
sgerqf(a,[lwork,overwrite_a])	Wrapper for sgerqf.
dgerqf(a,[lwork,overwrite_a])	Wrapper for dgerqf.
cgerqf(a,[lwork,overwrite_a])	Wrapper for cgerqf.
zgerqf(a,[lwork,overwrite_a])	Wrapper for zgerqf.
sgesdd(...)	Wrapper for sgesdd.
dgesdd(...)	Wrapper for dgesdd.
cgesdd(...)	Wrapper for cgesdd.
zgesdd(...)	Wrapper for zgesdd.
sgesdd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for sgesdd_lwork.
dgesdd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for dgesdd_lwork.
cgesdd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for cgesdd_lwork.
zgesdd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for zgesdd_lwork.
sgesvd(...)	Wrapper for sgesvd.
dgesvd(...)	Wrapper for dgesvd.
cgesvd(...)	Wrapper for cgesvd.
zgesvd(...)	Wrapper for zgesvd.
sgesvd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for sgesvd_lwork.
dgesvd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for dgesvd_lwork.
cgesvd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for cgesvd_lwork.
zgesvd_lwork(m,n,[compute_uv,full_matrices])	Wrapper for zgesvd_lwork.
sgesv(a,b,[overwrite_a,overwrite_b])	Wrapper for sgesv.
dgesv(a,b,[overwrite_a,overwrite_b])	Wrapper for dgesv.
cgesv(a,b,[overwrite_a,overwrite_b])	Wrapper for cgesv.
zgesv(a,b,[overwrite_a,overwrite_b])	Wrapper for zgesv.
sgesvx(...)	Wrapper for sgesvx.
dgesvx(...)	Wrapper for dgesvx.
cgesvx(...)	Wrapper for cgesvx.
zgesvx(...)	Wrapper for zgesvx.
sgecon(a,anorm,[norm])	Wrapper for sgecon.
dgecon(a,anorm,[norm])	Wrapper for dgecon.
cgecon(a,anorm,[norm])	Wrapper for cgecon.
zgecon(a,anorm,[norm])	Wrapper for zgecon.
ssysv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for ssysv.
dsysv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for dsysv.
csysv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for csysv.
zsysv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for zsysv.
ssysv_lwork(n,[lower])	Wrapper for ssysv_lwork.
dsysv_lwork(n,[lower])	Wrapper for dsysv_lwork.
csysv_lwork(n,[lower])	Wrapper for csysv_lwork.
zsysv_lwork(n,[lower])	Wrapper for zsysv_lwork.

Continued on next page

Table 5.85 – continued from previous page

ssysvx(...)	Wrapper for ssysvx.
dsysvx(...)	Wrapper for dsysvx.
csysvx(...)	Wrapper for csysvx.
zsysvx(...)	Wrapper for zsysvx.
ssysvx_lwork(n,[lower])	Wrapper for ssysvx_lwork.
dsysvx_lwork(n,[lower])	Wrapper for dsysvx_lwork.
csysvx_lwork(n,[lower])	Wrapper for csysvx_lwork.
zsysvx_lwork(n,[lower])	Wrapper for zsysvx_lwork.
chesv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for chesv.
zhesv(a,b,[lwork,lower,overwrite_a,overwrite_b])	Wrapper for zhesv.
chesv_lwork(n,[lower])	Wrapper for chesv_lwork.
zhesv_lwork(n,[lower])	Wrapper for zhesv_lwork.
chesvx(...)	Wrapper for chesvx.
zhesvx(...)	Wrapper for zhesvx.
chesvx_lwork(n,[lower])	Wrapper for chesvx_lwork.
zhesvx_lwork(n,[lower])	Wrapper for zhesvx_lwork.
sgetrf(a,[overwrite_a])	Wrapper for sgetrf.
dgetrf(a,[overwrite_a])	Wrapper for dgetrf.
cgetrf(a,[overwrite_a])	Wrapper for cgetrf.
zgetrf(a,[overwrite_a])	Wrapper for zgetrf.
sgetri(lu,piv,[lwork,overwrite_lu])	Wrapper for sgetri.
dgetri(lu,piv,[lwork,overwrite_lu])	Wrapper for dgetri.
cgetri(lu,piv,[lwork,overwrite_lu])	Wrapper for cgetri.
zgetri(lu,piv,[lwork,overwrite_lu])	Wrapper for zgetri.
sgetri_lwork(n)	Wrapper for sgetri_lwork.
dgetri_lwork(n)	Wrapper for dgetri_lwork.
cgetri_lwork(n)	Wrapper for cgetri_lwork.
zgetri_lwork(n)	Wrapper for zgetri_lwork.
sgetrs(lu,piv,b,[trans,overwrite_b])	Wrapper for sgetrs.
dgetrs(lu,piv,b,[trans,overwrite_b])	Wrapper for dgetrs.
cgetrs(lu,piv,b,[trans,overwrite_b])	Wrapper for cgetrs.
zgetrs(lu,piv,b,[trans,overwrite_b])	Wrapper for zgetrs.
sgges(...)	Wrapper for sgges.
dgges(...)	Wrapper for dgges.
cgges(...)	Wrapper for cgges.
zgges(...)	Wrapper for zgges.
sggev(...)	Wrapper for sggev.
dggev(...)	Wrapper for dggev.
cggev(...)	Wrapper for cggev.
zggev(...)	Wrapper for zggev.
chbevd(...)	Wrapper for chbevd.
zhbevd(...)	Wrapper for zhbevd.
chbevx(...)	Wrapper for chbevx.
zhbevx(...)	Wrapper for zhbevx.
cheev(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for cheev.
zheev(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for zheev.
cheevd(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for cheevd.
zheevd(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for zheevd.
cheevr(...)	Wrapper for cheevr.
zheevr(...)	Wrapper for zheevr.

Continued on next page

Table 5.85 – continued from previous page

chegv(...)	Wrapper for `chegv`.
zhegv(...)	Wrapper for `zhegv`.
chegvd(...)	Wrapper for `chegvd`.
zhegvd(...)	Wrapper for `zhegvd`.
chegvx(...)	Wrapper for `chegvx`.
zhegvx(...)	Wrapper for `zhegvx`.
slarf(v,tau,c,work,[side,incv,overwrite_c])	Wrapper for `slarf`.
dlarf(v,tau,c,work,[side,incv,overwrite_c])	Wrapper for `dlarf`.
clarf(v,tau,c,work,[side,incv,overwrite_c])	Wrapper for `clarf`.
zlarf(v,tau,c,work,[side,incv,overwrite_c])	Wrapper for `zlarf`.
slarfg(n,alpha,x,[incx,overwrite_x])	Wrapper for `slarfg`.
dlarfg(n,alpha,x,[incx,overwrite_x])	Wrapper for `dlarfg`.
clarfg(n,alpha,x,[incx,overwrite_x])	Wrapper for `clarfg`.
zlarfg(n,alpha,x,[incx,overwrite_x])	Wrapper for `zlarfg`.
slartg(f,g)	Wrapper for `slartg`.
dlartg(f,g)	Wrapper for `dlartg`.
clartg(f,g)	Wrapper for `clartg`.
zlartg(f,g)	Wrapper for `zlartg`.
slasd4(i,d,z,[rho])	Wrapper for `slasd4`.
dlasd4(i,d,z,[rho])	Wrapper for `dlasd4`.
slaswp(a,piv,[k1,k2,off,inc,overwrite_a])	Wrapper for `slaswp`.
dlaswp(a,piv,[k1,k2,off,inc,overwrite_a])	Wrapper for `dlaswp`.
claswp(a,piv,[k1,k2,off,inc,overwrite_a])	Wrapper for `claswp`.
zlaswp(a,piv,[k1,k2,off,inc,overwrite_a])	Wrapper for `zlaswp`.
slauum(c,[lower,overwrite_c])	Wrapper for `slauum`.
dlauum(c,[lower,overwrite_c])	Wrapper for `dlauum`.
clauum(c,[lower,overwrite_c])	Wrapper for `clauum`.
zlauum(c,[lower,overwrite_c])	Wrapper for `zlauum`.
spbsv(ab,b,[lower,ldab,overwrite_ab,overwrite_b])	Wrapper for `spbsv`.
dpbsv(ab,b,[lower,ldab,overwrite_ab,overwrite_b])	Wrapper for `dpbsv`.
cpbsv(ab,b,[lower,ldab,overwrite_ab,overwrite_b])	Wrapper for `cpbsv`.
zpbsv(ab,b,[lower,ldab,overwrite_ab,overwrite_b])	Wrapper for `zpbsv`.
spbtrf(ab,[lower,ldab,overwrite_ab])	Wrapper for `spbtrf`.
dpbtrf(ab,[lower,ldab,overwrite_ab])	Wrapper for `dpbtrf`.
cpbtrf(ab,[lower,ldab,overwrite_ab])	Wrapper for `cpbtrf`.
zpbtrf(ab,[lower,ldab,overwrite_ab])	Wrapper for `zpbtrf`.
spbtrs(ab,b,[lower,ldab,overwrite_b])	Wrapper for `spbtrs`.
dpbtrs(ab,b,[lower,ldab,overwrite_b])	Wrapper for `dpbtrs`.
cpbtrs(ab,b,[lower,ldab,overwrite_b])	Wrapper for `cpbtrs`.
zpbtrs(ab,b,[lower,ldab,overwrite_b])	Wrapper for `zpbtrs`.
sposv(a,b,[lower,overwrite_a,overwrite_b])	Wrapper for `sposv`.
dposv(a,b,[lower,overwrite_a,overwrite_b])	Wrapper for `dposv`.
cposv(a,b,[lower,overwrite_a,overwrite_b])	Wrapper for `cposv`.
zposv(a,b,[lower,overwrite_a,overwrite_b])	Wrapper for `zposv`.
sposvx(...)	Wrapper for `sposvx`.
dposvx(...)	Wrapper for `dposvx`.
cposvx(...)	Wrapper for `cposvx`.
zposvx(...)	Wrapper for `zposvx`.
spocon(a,anorm,[uplo])	Wrapper for `spocon`.
dpocon(a,anorm,[uplo])	Wrapper for `dpocon`.

Continued on next page

Table 5.85 – continued from previous page

cpocon(a,anorm,[uplo])	Wrapper for `cpocon`.
zpocon(a,anorm,[uplo])	Wrapper for `zpocon`.
spotrf(a,[lower,clean,overwrite_a])	Wrapper for `spotrf`.
dpotrf(a,[lower,clean,overwrite_a])	Wrapper for `dpotrf`.
cpotrf(a,[lower,clean,overwrite_a])	Wrapper for `cpotrf`.
zpotrf(a,[lower,clean,overwrite_a])	Wrapper for `zpotrf`.
spotri(c,[lower,overwrite_c])	Wrapper for `spotri`.
dpotri(c,[lower,overwrite_c])	Wrapper for `dpotri`.
cpotri(c,[lower,overwrite_c])	Wrapper for `cpotri`.
zpotri(c,[lower,overwrite_c])	Wrapper for `zpotri`.
spotrs(c,b,[lower,overwrite_b])	Wrapper for `spotrs`.
dpotrs(c,b,[lower,overwrite_b])	Wrapper for `dpotrs`.
cpotrs(c,b,[lower,overwrite_b])	Wrapper for `cpotrs`.
zpotrs(c,b,[lower,overwrite_b])	Wrapper for `zpotrs`.
crot(...)	Wrapper for `crot`.
zrot(...)	Wrapper for `zrot`.
strsyl(a,b,c,[trana,tranb,isgn,overwrite_c])	Wrapper for `strsyl`.
dtrsyl(a,b,c,[trana,tranb,isgn,overwrite_c])	Wrapper for `dtrsyl`.
ctrsyl(a,b,c,[trana,tranb,isgn,overwrite_c])	Wrapper for `ctrsyl`.
ztrsyl(a,b,c,[trana,tranb,isgn,overwrite_c])	Wrapper for `ztrsyl`.
strtri(c,[lower,unitdiag,overwrite_c])	Wrapper for `strtri`.
dtrtri(c,[lower,unitdiag,overwrite_c])	Wrapper for `dtrtri`.
ctrtri(c,[lower,unitdiag,overwrite_c])	Wrapper for `ctrtri`.
ztrtri(c,[lower,unitdiag,overwrite_c])	Wrapper for `ztrtri`.
strtrs(...)	Wrapper for `strtrs`.
dtrtrs(...)	Wrapper for `dtrtrs`.
ctrtrs(...)	Wrapper for `ctrtrs`.
ztrtrs(...)	Wrapper for `ztrtrs`.
cunghr(a,tau,[lo,hi,lwork,overwrite_a])	Wrapper for `cunghr`.
zunghr(a,tau,[lo,hi,lwork,overwrite_a])	Wrapper for `zunghr`.
cungqr(a,tau,[lwork,overwrite_a])	Wrapper for `cungqr`.
zungqr(a,tau,[lwork,overwrite_a])	Wrapper for `zungqr`.
cungrq(a,tau,[lwork,overwrite_a])	Wrapper for `cungrq`.
zungrq(a,tau,[lwork,overwrite_a])	Wrapper for `zungrq`.
cunmqr(side,trans,a,tau,c,lwork,[overwrite_c])	Wrapper for `cunmqr`.
zunmqr(side,trans,a,tau,c,lwork,[overwrite_c])	Wrapper for `zunmqr`.
sgtsv(...)	Wrapper for `sgtsv`.
dgtsv(...)	Wrapper for `dgtsv`.
cgtsv(...)	Wrapper for `cgtsv`.
zgtsv(...)	Wrapper for `zgtsv`.
sptsv(...)	Wrapper for `sptsv`.
dptsv(...)	Wrapper for `dptsv`.
cptsv(...)	Wrapper for `cptsv`.
zptsv(...)	Wrapper for `zptsv`.
slamch(cmach)	Wrapper for `slamch`.
dlamch(cmach)	Wrapper for `dlamch`.
sorghr(a,tau,[lo,hi,lwork,overwrite_a])	Wrapper for `sorghr`.
dorghr(a,tau,[lo,hi,lwork,overwrite_a])	Wrapper for `dorghr`.
sorgqr(a,tau,[lwork,overwrite_a])	Wrapper for `sorgqr`.
dorgqr(a,tau,[lwork,overwrite_a])	Wrapper for `dorgqr`.

Continued on next page

Table 5.85 – continued from previous page

sorgrq(a,tau,[lwork,overwrite_a])	Wrapper for `sorgrq`.
dorgrq(a,tau,[lwork,overwrite_a])	Wrapper for `dorgrq`.
sormqr(side,trans,a,tau,c,lwork,[overwrite_c])	Wrapper for `sormqr`.
dormqr(side,trans,a,tau,c,lwork,[overwrite_c])	Wrapper for `dormqr`.
ssbev(ab,[compute_v,lower,ldab,overwrite_ab])	Wrapper for `ssbev`.
dsbev(ab,[compute_v,lower,ldab,overwrite_ab])	Wrapper for `dsbev`.
ssbevd(...)	Wrapper for `ssbevd`.
dsbevd(...)	Wrapper for `dsbevd`.
ssbevx(...)	Wrapper for `ssbevx`.
dsbevx(...)	Wrapper for `dsbevx`.
ssyev(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for `ssyev`.
dsyev(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for `dsyev`.
ssyevd(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for `ssyevd`.
dsyevd(a,[compute_v,lower,lwork,overwrite_a])	Wrapper for `dsyevd`.
ssyevr(...)	Wrapper for `ssyevr`.
dsyevr(...)	Wrapper for `dsyevr`.
ssygv(...)	Wrapper for `ssygv`.
dsygv(...)	Wrapper for `dsygv`.
ssygvd(...)	Wrapper for `ssygvd`.
dsygvd(...)	Wrapper for `dsygvd`.
ssygvx(...)	Wrapper for `ssygvx`.
dsygvx(...)	Wrapper for `dsygvx`.
slange(norm,a)	Wrapper for `slange`.
dlange(norm,a)	Wrapper for `dlange`.
clange(norm,a)	Wrapper for `clange`.
zlange(norm,a)	Wrapper for `zlange`.
ilaver()	Wrapper for `ilaver`.

`scipy.linalg.lapack.`**`sgbsv`**(*kl*, *ku*, *ab*, *b*[, *overwrite_ab*, *overwrite_b*]) = <fortran object>

Wrapper for `sgbsv`.

Parameters
kl : input int
ku : input int
ab : input rank-2 array('f') with bounds (2*kl+ku+1,n)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns
lub : rank-2 array('f') with bounds (2*kl+ku+1,n) and ab storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dgbsv`**(*kl*, *ku*, *ab*, *b*[, *overwrite_ab*, *overwrite_b*]) = <fortran object>

Wrapper for `dgbsv`.

Parameters
kl : input int
ku : input int
ab : input rank-2 array('d') with bounds (2*kl+ku+1,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **lub** : rank-2 array('d') with bounds (2*kl+ku+1,n) and ab storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**cgbsv**(*kl*, *ku*, *ab*, *b*[, *overwrite_ab*, *overwrite_b*]) **= <fortran object>**
Wrapper for `cgbsv`.

Parameters **kl** : input int
ku : input int
ab : input rank-2 array('F') with bounds (2*kl+ku+1,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **lub** : rank-2 array('F') with bounds (2*kl+ku+1,n) and ab storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**zgbsv**(*kl*, *ku*, *ab*, *b*[, *overwrite_ab*, *overwrite_b*]) **= <fortran object>**
Wrapper for `zgbsv`.

Parameters **kl** : input int
ku : input int
ab : input rank-2 array('D') with bounds (2*kl+ku+1,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **lub** : rank-2 array('D') with bounds (2*kl+ku+1,n) and ab storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**sgbtrf**(*ab*, *kl*, *ku*[, *m*, *n*, *ldab*, *overwrite_ab*]) **= <fortran object>**
Wrapper for `sgbtrf`.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,*)
kl : input int
ku : input int

Returns **lu** : rank-2 array('f') with bounds (ldab,*) and ab storage
ipiv : rank-1 array('i') with bounds (MIN(m,n))
info : int

Other Parameters
m : input int, optional
Default: shape(ab,1)
n : input int, optional

Default: shape(ab,1)
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.dgbtrf`(*ab*, *kl*, *ku*[, *m*, *n*, *ldab*, *overwrite_ab*]) = <fortran object>
Wrapper for `dgbtrf`.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,*)
kl : input int
ku : input int
Returns **lu** : rank-2 array('d') with bounds (ldab,*) and ab storage
ipiv : rank-1 array('i') with bounds (MIN(m,n))
info : int
Other Parameters
m : input int, optional
Default: shape(ab,1)
n : input int, optional
Default: shape(ab,1)
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.cgbtrf`(*ab*, *kl*, *ku*[, *m*, *n*, *ldab*, *overwrite_ab*]) = <fortran object>
Wrapper for `cgbtrf`.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,*)
kl : input int
ku : input int
Returns **lu** : rank-2 array('F') with bounds (ldab,*) and ab storage
ipiv : rank-1 array('i') with bounds (MIN(m,n))
info : int
Other Parameters
m : input int, optional
Default: shape(ab,1)
n : input int, optional
Default: shape(ab,1)
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.zgbtrf`(*ab*, *kl*, *ku*[, *m*, *n*, *ldab*, *overwrite_ab*]) = <fortran object>
Wrapper for `zgbtrf`.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,*)
kl : input int
ku : input int
Returns **lu** : rank-2 array('D') with bounds (ldab,*) and ab storage
ipiv : rank-1 array('i') with bounds (MIN(m,n))
info : int
Other Parameters
m : input int, optional
Default: shape(ab,1)
n : input int, optional

Default: shape(ab,1)

overwrite_ab : input int, optional

Default: 0

ldab : input int, optional

Default: shape(ab,0)

scipy.linalg.lapack.**sgbtrs**(*ab*, *kl*, *ku*, *b*, *ipiv*[, *trans*, *n*, *ldab*, *ldb*, *overwrite_b*]) = **<fortran object>**

Wrapper for sgbtrs.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

kl : input int

ku : input int

b : input rank-2 array('f') with bounds (ldb,nrhs)

ipiv : input rank-1 array('i') with bounds (n)

Returns **x** : rank-2 array('f') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

overwrite_b : input int, optional

Default: 0

trans : input int, optional

Default: 0

n : input int, optional

Default: shape(ab,1)

ldab : input int, optional

Default: shape(ab,0)

ldb : input int, optional

Default: shape(b,0)

scipy.linalg.lapack.**dgbtrs**(*ab*, *kl*, *ku*, *b*, *ipiv*[, *trans*, *n*, *ldab*, *ldb*, *overwrite_b*]) = **<fortran object>**

Wrapper for dgbtrs.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)

kl : input int

ku : input int

b : input rank-2 array('d') with bounds (ldb,nrhs)

ipiv : input rank-1 array('i') with bounds (n)

Returns **x** : rank-2 array('d') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

overwrite_b : input int, optional

Default: 0

trans : input int, optional

Default: 0

n : input int, optional

Default: shape(ab,1)

ldab : input int, optional

Default: shape(ab,0)

ldb : input int, optional

Default: shape(b,0)

scipy.linalg.lapack.**cgbtrs**(*ab*, *kl*, *ku*, *b*, *ipiv*[, *trans*, *n*, *ldab*, *ldb*, *overwrite_b*]) = **<fortran object>**

Wrapper for cgbtrs.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,n)
kl : input int
ku : input int
b : input rank-2 array('F') with bounds (ldb,nrhs)
ipiv : input rank-1 array('i') with bounds (n)

Returns **x** : rank-2 array('F') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0
n : input int, optional
Default: shape(ab,1)
ldab : input int, optional
Default: shape(ab,0)
ldb : input int, optional
Default: shape(b,0)

scipy.linalg.lapack.**zgbtrs**(*ab*, *kl*, *ku*, *b*, *ipiv*[, *trans*, *n*, *ldab*, *ldb*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zgbtrs`.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,n)
kl : input int
ku : input int
b : input rank-2 array('D') with bounds (ldb,nrhs)
ipiv : input rank-1 array('i') with bounds (n)

Returns **x** : rank-2 array('D') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0
n : input int, optional
Default: shape(ab,1)
ldab : input int, optional
Default: shape(ab,0)
ldb : input int, optional
Default: shape(b,0)

scipy.linalg.lapack.**sgebal**(*a*[, *scale*, *permute*, *overwrite_a*]) **= <fortran object>**

Wrapper for `sgebal`.

Parameters **a** : input rank-2 array('f') with bounds (m,n)

Returns **ba** : rank-2 array('f') with bounds (m,n) and a storage
lo : int
hi : int
pivscale : rank-1 array('f') with bounds (n)
info : int

Other Parameters
scale : input int, optional
Default: 0
permute : input int, optional
Default: 0

overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**dgebal**(*a*[, *scale*, *permute*, *overwrite_a*]) = <fortran object>
Wrapper for dgebal.

Parameters **a** : input rank-2 array('d') with bounds (m,n)

Returns **ba** : rank-2 array('d') with bounds (m,n) and a storage
lo : int
hi : int
pivscale : rank-1 array('d') with bounds (n)
info : int

Other Parameters
scale : input int, optional
Default: 0
permute : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**cgebal**(*a*[, *scale*, *permute*, *overwrite_a*]) = <fortran object>
Wrapper for cgebal.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **ba** : rank-2 array('F') with bounds (m,n) and a storage
lo : int
hi : int
pivscale : rank-1 array('f') with bounds (n)
info : int

Other Parameters
scale : input int, optional
Default: 0
permute : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**zgebal**(*a*[, *scale*, *permute*, *overwrite_a*]) = <fortran object>
Wrapper for zgebal.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **ba** : rank-2 array('D') with bounds (m,n) and a storage
lo : int
hi : int
pivscale : rank-1 array('d') with bounds (n)
info : int

Other Parameters
scale : input int, optional
Default: 0
permute : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**sgees**(*sselect*, *a*[, *compute_v*, *sort_t*, *lwork*, *sselect_extra_args*, *overwrite_a*]) = <fortran object>
Wrapper for sgees.

Parameters
sselect : call-back function
a : input rank-2 array('f') with bounds (n,n)

Returns
t : rank-2 array('f') with bounds (n,n) and a storage
sdim : int
wr : rank-1 array('f') with bounds (n)
wi : rank-1 array('f') with bounds (n)
vs : rank-2 array('f') with bounds (ldvs,n)
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int

Other Parameters
compute_v : input int, optional
Default: 1
sort_t : input int, optional
Default: 0
sselect_extra_args : input tuple, optional
Default: ()
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

Notes

Call-back functions:

```
def sselect(arg1,arg2): return sselect
Required arguments:
  arg1 : input float
  arg2 : input float
Return objects:
  sselect : int
```

`scipy.linalg.lapack.`**`dgees`**(*dselect*, *a*[, *compute_v*, *sort_t*, *lwork*, *dselect_extra_args*, *overwrite_a*]) = **<fortran object>**

Wrapper for `dgees`.

Parameters
dselect : call-back function
a : input rank-2 array('d') with bounds (n,n)

Returns
t : rank-2 array('d') with bounds (n,n) and a storage
sdim : int
wr : rank-1 array('d') with bounds (n)
wi : rank-1 array('d') with bounds (n)
vs : rank-2 array('d') with bounds (ldvs,n)
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int

Other Parameters
compute_v : input int, optional
Default: 1
sort_t : input int, optional
Default: 0
dselect_extra_args : input tuple, optional
Default: ()
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

Notes

Call-back functions:

```
def dselect(arg1,arg2): return dselect
Required arguments:
  arg1 : input float
  arg2 : input float
Return objects:
  dselect : int
```

scipy.linalg.lapack.**cgees**(*cselect*, *a*[, *compute_v*, *sort_t*, *lwork*, *cselect_extra_args*, *overwrite_a*]) = <fortran object>

Wrapper for cgees.

Parameters
- **cselect** : call-back function
- **a** : input rank-2 array('F') with bounds (n,n)

Returns
- **t** : rank-2 array('F') with bounds (n,n) and a storage
- **sdim** : int
- **w** : rank-1 array('F') with bounds (n)
- **vs** : rank-2 array('F') with bounds (ldvs,n)
- **work** : rank-1 array('F') with bounds (MAX(lwork,1))
- **info** : int

Other Parameters
- **compute_v** : input int, optional
  Default: 1
- **sort_t** : input int, optional
  Default: 0
- **cselect_extra_args** : input tuple, optional
  Default: ()
- **overwrite_a** : input int, optional
  Default: 0
- **lwork** : input int, optional
  Default: 3*n

Notes

Call-back functions:

```
def cselect(arg): return cselect
Required arguments:
  arg : input complex
Return objects:
  cselect : int
```

scipy.linalg.lapack.**zgees**(*zselect*, *a*[, *compute_v*, *sort_t*, *lwork*, *zselect_extra_args*, *overwrite_a*]) = <fortran object>

Wrapper for zgees.

Parameters
- **zselect** : call-back function
- **a** : input rank-2 array('D') with bounds (n,n)

Returns
- **t** : rank-2 array('D') with bounds (n,n) and a storage
- **sdim** : int
- **w** : rank-1 array('D') with bounds (n)
- **vs** : rank-2 array('D') with bounds (ldvs,n)
- **work** : rank-1 array('D') with bounds (MAX(lwork,1))
- **info** : int

Other Parameters

compute_v : input int, optional
Default: 1

sort_t : input int, optional
Default: 0

zselect_extra_args : input tuple, optional
Default: ()

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 3*n

Notes

Call-back functions:

```
def zselect(arg): return zselect
Required arguments:
  arg : input complex
Return objects:
  zselect : int
```

`scipy.linalg.lapack.`**`sgeev`**(*a*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `sgeev`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)

Returns **wr** : rank-1 array('f') with bounds (n)

wi : rank-1 array('f') with bounds (n)

vl : rank-2 array('f') with bounds (ldvl,n)

vr : rank-2 array('f') with bounds (ldvr,n)

info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 4*n

`scipy.linalg.lapack.`**`dgeev`**(*a*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `dgeev`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)

Returns **wr** : rank-1 array('d') with bounds (n)

wi : rank-1 array('d') with bounds (n)

vl : rank-2 array('d') with bounds (ldvl,n)

vr : rank-2 array('d') with bounds (ldvr,n)

info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 4*n

`scipy.linalg.lapack.`**`cgeev`**(*a*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `cgeev`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)

Returns **w** : rank-1 array('F') with bounds (n)
vl : rank-2 array('F') with bounds (ldvl,n)
vr : rank-2 array('F') with bounds (ldvr,n)
info : int

Other Parameters
compute_vl : input int, optional
Default: 1
compute_vr : input int, optional
Default: 1
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n

`scipy.linalg.lapack.`**`zgeev`**(*a*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zgeev`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)

Returns **w** : rank-1 array('D') with bounds (n)
vl : rank-2 array('D') with bounds (ldvl,n)
vr : rank-2 array('D') with bounds (ldvr,n)
info : int

Other Parameters
compute_vl : input int, optional
Default: 1
compute_vr : input int, optional
Default: 1
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n

`scipy.linalg.lapack.`**`sgeev_lwork`**(*n*[, *compute_vl*, *compute_vr*]) **= <fortran object>**
Wrapper for `sgeev_lwork`.

Parameters **n** : input int

Returns **work** : float
info : int

Other Parameters
compute_vl : input int, optional
Default: 1
compute_vr : input int, optional
Default: 1

`scipy.linalg.lapack.`**`dgeev_lwork`**(*n*[, *compute_vl*, *compute_vr*]) **= <fortran object>**
Wrapper for `dgeev_lwork`.

Parameters **n** : input int

Returns **work** : float
info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

scipy.linalg.lapack.**cgeev_lwork**(*n*[, *compute_vl*, *compute_vr*]) = <fortran object>

Wrapper for `cgeev_lwork`.

Parameters **n** : input int

Returns **work** : complex
info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

scipy.linalg.lapack.**zgeev_lwork**(*n*[, *compute_vl*, *compute_vr*]) = <fortran object>

Wrapper for `zgeev_lwork`.

Parameters **n** : input int

Returns **work** : complex
info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

scipy.linalg.lapack.**sgegv**(**args*, ***kwds*)

sgegv is deprecated! The **gegv* family of routines has been deprecated in LAPACK 3.6.0 in favor of the **ggev* family of routines. The corresponding wrappers will be removed from SciPy in a future release.

alphar,alphai,beta,vl,vr,info = sgegv(a,b,[compute_vl,compute_vr,lwork,overwrite_a,overwrite_b])

Wrapper for `sgegv`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,n)

Returns **alphar** : rank-1 array('f') with bounds (n)
alphai : rank-1 array('f') with bounds (n)
beta : rank-1 array('f') with bounds (n)
vl : rank-2 array('f') with bounds (ldvl,n)
vr : rank-2 array('f') with bounds (ldvr,n)
info : int

Other Parameters

compute_vl : input int, optional
Default: 1

compute_vr : input int, optional
Default: 1

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

lwork : input int, optional
Default: 8*n

`scipy.linalg.lapack.dgegv(*args, **kwds)`

dgegv is deprecated! The **gegv* family of routines has been deprecated in LAPACK 3.6.0 in favor of the **ggev* family of routines. The corresponding wrappers will be removed from SciPy in a future release.

alphar,alphai,beta,vl,vr,info = dgegv(a,b,[compute_vl,compute_vr,lwork,overwrite_a,overwrite_b])

Wrapper for `dgegv`.

Parameters
- **a** : input rank-2 array('d') with bounds (n,n)
- **b** : input rank-2 array('d') with bounds (n,n)

Returns
- **alphar** : rank-1 array('d') with bounds (n)
- **alphai** : rank-1 array('d') with bounds (n)
- **beta** : rank-1 array('d') with bounds (n)
- **vl** : rank-2 array('d') with bounds (ldvl,n)
- **vr** : rank-2 array('d') with bounds (ldvr,n)
- **info** : int

Other Parameters
- **compute_vl** : input int, optional
  Default: 1
- **compute_vr** : input int, optional
  Default: 1
- **overwrite_a** : input int, optional
  Default: 0
- **overwrite_b** : input int, optional
  Default: 0
- **lwork** : input int, optional
  Default: 8*n

`scipy.linalg.lapack.cgegv(*args, **kwds)`

cgegv is deprecated! The **gegv* family of routines has been deprecated in LAPACK 3.6.0 in favor of the **ggev* family of routines. The corresponding wrappers will be removed from SciPy in a future release.

alpha,beta,vl,vr,info = cgegv(a,b,[compute_vl,compute_vr,lwork,overwrite_a,overwrite_b])

Wrapper for `cgegv`.

Parameters
- **a** : input rank-2 array('F') with bounds (n,n)
- **b** : input rank-2 array('F') with bounds (n,n)

Returns
- **alpha** : rank-1 array('F') with bounds (n)
- **beta** : rank-1 array('F') with bounds (n)
- **vl** : rank-2 array('F') with bounds (ldvl,n)
- **vr** : rank-2 array('F') with bounds (ldvr,n)
- **info** : int

Other Parameters
- **compute_vl** : input int, optional
  Default: 1
- **compute_vr** : input int, optional
  Default: 1
- **overwrite_a** : input int, optional
  Default: 0
- **overwrite_b** : input int, optional
  Default: 0
- **lwork** : input int, optional
  Default: 2*n

`scipy.linalg.lapack.zgegv(*args, **kwds)`

zgegv is deprecated! The **gegv* family of routines has been deprecated in LAPACK 3.6.0 in favor of the **ggev* family of routines. The corresponding wrappers will be removed from SciPy in a future release.

alpha,beta,vl,vr,info = zgegv(a,b,[compute_vl,compute_vr,lwork,overwrite_a,overwrite_b])

Wrapper for `zgegv`.

Parameters
: **a** : input rank-2 array('D') with bounds (n,n)
: **b** : input rank-2 array('D') with bounds (n,n)

Returns
: **alpha** : rank-1 array('D') with bounds (n)
: **beta** : rank-1 array('D') with bounds (n)
: **vl** : rank-2 array('D') with bounds (ldvl,n)
: **vr** : rank-2 array('D') with bounds (ldvr,n)
: **info** : int

Other Parameters
: **compute_vl** : input int, optional
    Default: 1
: **compute_vr** : input int, optional
    Default: 1
: **overwrite_a** : input int, optional
    Default: 0
: **overwrite_b** : input int, optional
    Default: 0
: **lwork** : input int, optional
    Default: 2*n

`scipy.linalg.lapack.`**`sgehrd`**(*a*[, *lo*, *hi*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `sgehrd`.

Parameters
: **a** : input rank-2 array('f') with bounds (n,n)

Returns
: **ht** : rank-2 array('f') with bounds (n,n) and a storage
: **tau** : rank-1 array('f') with bounds (n - 1)
: **info** : int

Other Parameters
: **lo** : input int, optional
    Default: 0
: **hi** : input int, optional
    Default: n-1
: **overwrite_a** : input int, optional
    Default: 0
: **lwork** : input int, optional
    Default: MAX(n,1)

`scipy.linalg.lapack.`**`dgehrd`**(*a*[, *lo*, *hi*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `dgehrd`.

Parameters
: **a** : input rank-2 array('d') with bounds (n,n)

Returns
: **ht** : rank-2 array('d') with bounds (n,n) and a storage
: **tau** : rank-1 array('d') with bounds (n - 1)
: **info** : int

Other Parameters
: **lo** : input int, optional
    Default: 0
: **hi** : input int, optional
    Default: n-1
: **overwrite_a** : input int, optional
    Default: 0
: **lwork** : input int, optional
    Default: MAX(n,1)

scipy.linalg.lapack.**cgehrd**(*a*[, *lo*, *hi*, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for `cgehrd`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)

Returns **ht** : rank-2 array('F') with bounds (n,n) and a storage

tau : rank-1 array('F') with bounds (n - 1)

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: MAX(n,1)

scipy.linalg.lapack.**zgehrd**(*a*[, *lo*, *hi*, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for `zgehrd`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)

Returns **ht** : rank-2 array('D') with bounds (n,n) and a storage

tau : rank-1 array('D') with bounds (n - 1)

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: MAX(n,1)

scipy.linalg.lapack.**sgehrd_lwork**(*n*[, *lo*, *hi*]) = <fortran object>

Wrapper for `sgehrd_lwork`.

Parameters **n** : input int

Returns **work** : float

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

scipy.linalg.lapack.**dgehrd_lwork**(*n*[, *lo*, *hi*]) = <fortran object>

Wrapper for `dgehrd_lwork`.

Parameters **n** : input int

Returns **work** : float

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

scipy.linalg.lapack.**cgehrd_lwork**(*n*[, *lo*, *hi*]) = <fortran object>
Wrapper for cgehrd_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lo : input int, optional
Default: 0
hi : input int, optional
Default: n-1

scipy.linalg.lapack.**zgehrd_lwork**(*n*[, *lo*, *hi*]) = <fortran object>
Wrapper for zgehrd_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lo : input int, optional
Default: 0
hi : input int, optional
Default: n-1

scipy.linalg.lapack.**sgelss**(*a*, *b*[, *cond*, *lwork*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for sgelss.

Parameters **a** : input rank-2 array('f') with bounds (m,n)
b : input rank-2 array('f') with bounds (maxmn,nrhs)
Returns **v** : rank-2 array('f') with bounds (m,n) and a storage
x : rank-2 array('f') with bounds (maxmn,nrhs) and b storage
s : rank-1 array('f') with bounds (minmn)
rank : int
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: 3*minmn+MAX(2*minmn,MAX(maxmn,nrhs))

scipy.linalg.lapack.**dgelss**(*a*, *b*[, *cond*, *lwork*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for dgelss.

Parameters **a** : input rank-2 array('d') with bounds (m,n)
b : input rank-2 array('d') with bounds (maxmn,nrhs)
Returns **v** : rank-2 array('d') with bounds (m,n) and a storage
x : rank-2 array('d') with bounds (maxmn,nrhs) and b storage
s : rank-1 array('d') with bounds (minmn)
rank : int
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

cond : input float, optional
Default: -1.0

lwork : input int, optional
Default: 3*minmn+MAX(2*minmn,MAX(maxmn,nrhs))

`scipy.linalg.lapack.`**`cgelss`**(*a*, *b*[, *cond*, *lwork*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `cgelss`.

Parameters

a : input rank-2 array('F') with bounds (m,n)

b : input rank-2 array('F') with bounds (maxmn,nrhs)

Returns

v : rank-2 array('F') with bounds (m,n) and a storage

x : rank-2 array('F') with bounds (maxmn,nrhs) and b storage

s : rank-1 array('f') with bounds (minmn)

rank : int

work : rank-1 array('F') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

cond : input float, optional
Default: -1.0

lwork : input int, optional
Default: 2*minmn+MAX(maxmn,nrhs)

`scipy.linalg.lapack.`**`zgelss`**(*a*, *b*[, *cond*, *lwork*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zgelss`.

Parameters

a : input rank-2 array('D') with bounds (m,n)

b : input rank-2 array('D') with bounds (maxmn,nrhs)

Returns

v : rank-2 array('D') with bounds (m,n) and a storage

x : rank-2 array('D') with bounds (maxmn,nrhs) and b storage

s : rank-1 array('d') with bounds (minmn)

rank : int

work : rank-1 array('D') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

cond : input float, optional
Default: -1.0

lwork : input int, optional
Default: 2*minmn+MAX(maxmn,nrhs)

`scipy.linalg.lapack.`**`sgelss_lwork`**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) **= <fortran object>**

Wrapper for `sgelss_lwork`.

Parameters **m** : input int
n : input int
nrhs : input int
Returns **work** : float
info : int
Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

scipy.linalg.lapack.**dgelss_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>
Wrapper for dgelss_lwork.

Parameters **m** : input int
n : input int
nrhs : input int
Returns **work** : float
info : int
Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

scipy.linalg.lapack.**cgelss_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>
Wrapper for cgelss_lwork.

Parameters **m** : input int
n : input int
nrhs : input int
Returns **work** : complex
info : int
Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

scipy.linalg.lapack.**zgelss_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>
Wrapper for zgelss_lwork.

Parameters **m** : input int
n : input int
nrhs : input int
Returns **work** : complex
info : int
Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

scipy.linalg.lapack.**sgelsd**(*a*, *b*, *lwork*, *size_iwork*[, *cond*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for sgelsd.

Parameters
a : input rank-2 array('f') with bounds (m,n)
b : input rank-2 array('f') with bounds (maxmn,nrhs)
lwork : input int
size_iwork : input int

Returns
x : rank-2 array('f') with bounds (maxmn,nrhs) and b storage
s : rank-1 array('f') with bounds (minmn)
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
cond : input float, optional
Default: -1.0

scipy.linalg.lapack.**dgelsd**(*a*, *b*, *lwork*, *size_iwork*[, *cond*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dgelsd`.

Parameters
a : input rank-2 array('d') with bounds (m,n)
b : input rank-2 array('d') with bounds (maxmn,nrhs)
lwork : input int
size_iwork : input int

Returns
x : rank-2 array('d') with bounds (maxmn,nrhs) and b storage
s : rank-1 array('d') with bounds (minmn)
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
cond : input float, optional
Default: -1.0

scipy.linalg.lapack.**cgelsd**(*a*, *b*, *lwork*, *size_rwork*, *size_iwork*[, *cond*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `cgelsd`.

Parameters
a : input rank-2 array('F') with bounds (m,n)
b : input rank-2 array('F') with bounds (maxmn,nrhs)
lwork : input int
size_rwork : input int
size_iwork : input int

Returns
x : rank-2 array('F') with bounds (maxmn,nrhs) and b storage
s : rank-1 array('f') with bounds (minmn)
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
cond : input float, optional
Default: -1.0

scipy.linalg.lapack.**zgelsd**(*a*, *b*, *lwork*, *size_rwork*, *size_iwork*[, *cond*, *overwrite_a*, *overwrite_b*]) = <fortran object>

Wrapper for `zgelsd`.

Parameters
- **a** : input rank-2 array('D') with bounds (m,n)
- **b** : input rank-2 array('D') with bounds (maxmn,nrhs)
- **lwork** : input int
- **size_rwork** : input int
- **size_iwork** : input int

Returns
- **x** : rank-2 array('D') with bounds (maxmn,nrhs) and b storage
- **s** : rank-1 array('d') with bounds (minmn)
- **rank** : int
- **info** : int

Other Parameters
- **overwrite_a** : input int, optional
  Default: 0
- **overwrite_b** : input int, optional
  Default: 0
- **cond** : input float, optional
  Default: -1.0

scipy.linalg.lapack.**sgelsd_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>

Wrapper for `sgelsd_lwork`.

Parameters
- **m** : input int
- **n** : input int
- **nrhs** : input int

Returns
- **work** : float
- **iwork** : int
- **info** : int

Other Parameters
- **cond** : input float, optional
  Default: -1.0
- **lwork** : input int, optional
  Default: -1

scipy.linalg.lapack.**dgelsd_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>

Wrapper for `dgelsd_lwork`.

Parameters
- **m** : input int
- **n** : input int
- **nrhs** : input int

Returns
- **work** : float
- **iwork** : int
- **info** : int

Other Parameters
- **cond** : input float, optional
  Default: -1.0
- **lwork** : input int, optional
  Default: -1

scipy.linalg.lapack.**cgelsd_lwork**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) = <fortran object>

Wrapper for `cgelsd_lwork`.

Parameters
- **m** : input int
- **n** : input int
- **nrhs** : input int

Returns
work : complex
rwork : float
iwork : int
info : int

Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

`scipy.linalg.lapack.`**`zgelsd_lwork`**(*m*, *n*, *nrhs*[, *cond*, *lwork*]) **= <fortran object>**

Wrapper for `zgelsd_lwork`.

Parameters
m : input int
n : input int
nrhs : input int

Returns
work : complex
rwork : float
iwork : int
info : int

Other Parameters
cond : input float, optional
Default: -1.0
lwork : input int, optional
Default: -1

`scipy.linalg.lapack.`**`sgelsy`**(*a*, *b*, *jptv*, *cond*, *lwork*[, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `sgelsy`.

Parameters
a : input rank-2 array('f') with bounds (m,n)
b : input rank-2 array('f') with bounds (maxmn,nrhs)
jptv : input rank-1 array('i') with bounds (n)
cond : input float
lwork : input int

Returns
v : rank-2 array('f') with bounds (m,n) and a storage
x : rank-2 array('f') with bounds (maxmn,nrhs) and b storage
j : rank-1 array('i') with bounds (n) and jptv storage
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dgelsy`**(*a*, *b*, *jptv*, *cond*, *lwork*[, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dgelsy`.

Parameters
a : input rank-2 array('d') with bounds (m,n)
b : input rank-2 array('d') with bounds (maxmn,nrhs)
jptv : input rank-1 array('i') with bounds (n)
cond : input float
lwork : input int

Returns **v** : rank-2 array('d') with bounds (m,n) and a storage
x : rank-2 array('d') with bounds (maxmn,nrhs) and b storage
j : rank-1 array('i') with bounds (n) and jptv storage
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**cgelsy**(*a*, *b*, *jptv*, *cond*, *lwork*[, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for cgelsy.

Parameters **a** : input rank-2 array('F') with bounds (m,n)
b : input rank-2 array('F') with bounds (maxmn,nrhs)
jptv : input rank-1 array('i') with bounds (n)
cond : input float
lwork : input int

Returns **v** : rank-2 array('F') with bounds (m,n) and a storage
x : rank-2 array('F') with bounds (maxmn,nrhs) and b storage
j : rank-1 array('i') with bounds (n) and jptv storage
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**zgelsy**(*a*, *b*, *jptv*, *cond*, *lwork*[, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for zgelsy.

Parameters **a** : input rank-2 array('D') with bounds (m,n)
b : input rank-2 array('D') with bounds (maxmn,nrhs)
jptv : input rank-1 array('i') with bounds (n)
cond : input float
lwork : input int

Returns **v** : rank-2 array('D') with bounds (m,n) and a storage
x : rank-2 array('D') with bounds (maxmn,nrhs) and b storage
j : rank-1 array('i') with bounds (n) and jptv storage
rank : int
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**sgelsy_lwork**(*m*, *n*, *nrhs*, *cond*[, *lwork*]) = **<fortran object>**

Wrapper for sgelsy_lwork.

Parameters **m** : input int
n : input int
nrhs : input int

cond : input float

Returns **work** : float

info : int

Other Parameters

lwork : input int, optional

Default: -1

scipy.linalg.lapack.**dgelsy_lwork**(*m*, *n*, *nrhs*, *cond*[, *lwork*]) = <fortran object>

Wrapper for dgelsy_lwork.

Parameters **m** : input int

n : input int

nrhs : input int

cond : input float

Returns **work** : float

info : int

Other Parameters

lwork : input int, optional

Default: -1

scipy.linalg.lapack.**cgelsy_lwork**(*m*, *n*, *nrhs*, *cond*[, *lwork*]) = <fortran object>

Wrapper for cgelsy_lwork.

Parameters **m** : input int

n : input int

nrhs : input int

cond : input float

Returns **work** : complex

info : int

Other Parameters

lwork : input int, optional

Default: -1

scipy.linalg.lapack.**zgelsy_lwork**(*m*, *n*, *nrhs*, *cond*[, *lwork*]) = <fortran object>

Wrapper for zgelsy_lwork.

Parameters **m** : input int

n : input int

nrhs : input int

cond : input float

Returns **work** : complex

info : int

Other Parameters

lwork : input int, optional

Default: -1

scipy.linalg.lapack.**sgeqp3**(*a*[, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for sgeqp3.

Parameters **a** : input rank-2 array(‘f’) with bounds (m,n)

Returns **qr** : rank-2 array(‘f’) with bounds (m,n) and a storage

jpvt : rank-1 array(‘i’) with bounds (n)

tau : rank-1 array(‘f’) with bounds (MIN(m,n))

work : rank-1 array(‘f’) with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional

Default: 0

lwork : input int, optional
Default: 3*(n+1)

scipy.linalg.lapack.**dgeqp3**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for dgeqp3.

Parameters **a** : input rank-2 array('d') with bounds (m,n)

Returns **qr** : rank-2 array('d') with bounds (m,n) and a storage
jpvt : rank-1 array('i') with bounds (n)
tau : rank-1 array('d') with bounds (MIN(m,n))
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*(n+1)

scipy.linalg.lapack.**cgeqp3**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for cgeqp3.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **qr** : rank-2 array('F') with bounds (m,n) and a storage
jpvt : rank-1 array('i') with bounds (n)
tau : rank-1 array('F') with bounds (MIN(m,n))
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*(n+1)

scipy.linalg.lapack.**zgeqp3**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for zgeqp3.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **qr** : rank-2 array('D') with bounds (m,n) and a storage
jpvt : rank-1 array('i') with bounds (n)
tau : rank-1 array('D') with bounds (MIN(m,n))
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*(n+1)

scipy.linalg.lapack.**sgeqrf**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for sgeqrf.

Parameters **a** : input rank-2 array('f') with bounds (m,n)

Returns **qr** : rank-2 array('f') with bounds (m,n) and a storage
tau : rank-1 array('f') with bounds (MIN(m,n))
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional

Default: 0

lwork : input int, optional

Default: 3*n

scipy.linalg.lapack.**dgeqrf**(*a*[, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for dgeqrf.

Parameters **a** : input rank-2 array('d') with bounds (m,n)

Returns **qr** : rank-2 array('d') with bounds (m,n) and a storage

tau : rank-1 array('d') with bounds (MIN(m,n))

work : rank-1 array('d') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional

Default: 0

lwork : input int, optional

Default: 3*n

scipy.linalg.lapack.**cgeqrf**(*a*[, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for cgeqrf.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **qr** : rank-2 array('F') with bounds (m,n) and a storage

tau : rank-1 array('F') with bounds (MIN(m,n))

work : rank-1 array('F') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional

Default: 0

lwork : input int, optional

Default: 3*n

scipy.linalg.lapack.**zgeqrf**(*a*[, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for zgeqrf.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **qr** : rank-2 array('D') with bounds (m,n) and a storage

tau : rank-1 array('D') with bounds (MIN(m,n))

work : rank-1 array('D') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional

Default: 0

lwork : input int, optional

Default: 3*n

scipy.linalg.lapack.**sgerqf**(*a*[, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for sgerqf.

Parameters **a** : input rank-2 array('f') with bounds (m,n)

Returns **qr** : rank-2 array('f') with bounds (m,n) and a storage

tau : rank-1 array('f') with bounds (MIN(m,n))

work : rank-1 array('f') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional

Default: 0

lwork : input int, optional

Default: 3*m

scipy.linalg.lapack.**dgerqf**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for dgerqf.

Parameters **a** : input rank-2 array('d') with bounds (m,n)
Returns **qr** : rank-2 array('d') with bounds (m,n) and a storage
tau : rank-1 array('d') with bounds (MIN(m,n))
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

scipy.linalg.lapack.**cgerqf**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for cgerqf.

Parameters **a** : input rank-2 array('F') with bounds (m,n)
Returns **qr** : rank-2 array('F') with bounds (m,n) and a storage
tau : rank-1 array('F') with bounds (MIN(m,n))
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

scipy.linalg.lapack.**zgerqf**(*a*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for zgerqf.

Parameters **a** : input rank-2 array('D') with bounds (m,n)
Returns **qr** : rank-2 array('D') with bounds (m,n) and a storage
tau : rank-1 array('D') with bounds (MIN(m,n))
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

scipy.linalg.lapack.**sgesdd**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for sgesdd.

Parameters **a** : input rank-2 array('f') with bounds (m,n)
Returns **u** : rank-2 array('f') with bounds (u0,u1)
s : rank-1 array('f') with bounds (minmn)
vt : rank-2 array('f') with bounds (vt0,vt1)
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
compute_uv : input int, optional
Default: 1

full_matrices : input int, optional
Default: 1
lwork : input int, optional
Default: (compute_uv?4*minmn*minmn+MAX(m,n)+9*minmn:MAX(14*minmn+4,10*minmn+2+25*

`scipy.linalg.lapack.`**`dgesdd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) = **<fortran object>**

Wrapper for `dgesdd`.

Parameters **a** : input rank-2 array('d') with bounds (m,n)

Returns **u** : rank-2 array('d') with bounds (u0,u1)
s : rank-1 array('d') with bounds (minmn)
vt : rank-2 array('d') with bounds (vt0,vt1)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1
lwork : input int, optional
Default: (compute_uv?4*minmn*minmn+MAX(m,n)+9*minmn:MAX(14*minmn+4,10*minmn+2+25*

`scipy.linalg.lapack.`**`cgesdd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) = **<fortran object>**

Wrapper for `cgesdd`.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **u** : rank-2 array('F') with bounds (u0,u1)
s : rank-1 array('f') with bounds (minmn)
vt : rank-2 array('F') with bounds (vt0,vt1)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1
lwork : input int, optional
Default: (compute_uv?2*minmn*minmn+MAX(m,n)+2*minmn:2*minmn+MAX(m,n))

`scipy.linalg.lapack.`**`zgesdd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) = **<fortran object>**

Wrapper for `zgesdd`.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **u** : rank-2 array('D') with bounds (u0,u1)
s : rank-1 array('d') with bounds (minmn)
vt : rank-2 array('D') with bounds (vt0,vt1)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
compute_uv : input int, optional
Default: 1

full_matrices : input int, optional
Default: 1
lwork : input int, optional
Default: (compute_uv?2*minmn*minmn+MAX(m,n)+2*minmn:2*minmn+MAX(m,n))

scipy.linalg.lapack.**sgesdd_lwork**(*m*, *n*[, *compute_uv*, *full_matrices*]) = <fortran object>
Wrapper for sgesdd_lwork.

Parameters **m** : input int
n : input int
Returns **work** : float
info : int
Other Parameters
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1

scipy.linalg.lapack.**dgesdd_lwork**(*m*, *n*[, *compute_uv*, *full_matrices*]) = <fortran object>
Wrapper for dgesdd_lwork.

Parameters **m** : input int
n : input int
Returns **work** : float
info : int
Other Parameters
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1

scipy.linalg.lapack.**cgesdd_lwork**(*m*, *n*[, *compute_uv*, *full_matrices*]) = <fortran object>
Wrapper for cgesdd_lwork.

Parameters **m** : input int
n : input int
Returns **work** : complex
info : int
Other Parameters
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1

scipy.linalg.lapack.**zgesdd_lwork**(*m*, *n*[, *compute_uv*, *full_matrices*]) = <fortran object>
Wrapper for zgesdd_lwork.

Parameters **m** : input int
n : input int
Returns **work** : complex
info : int
Other Parameters
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1

`scipy.linalg.lapack.`**`sgesvd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `sgesvd`.

Parameters **a** : input rank-2 array('f') with bounds (m,n)

Returns **u** : rank-2 array('f') with bounds (u0,u1)

s : rank-1 array('f') with bounds (minmn)

vt : rank-2 array('f') with bounds (vt0,vt1)

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

compute_uv : input int, optional
Default: 1

full_matrices : input int, optional
Default: 1

lwork : input int, optional
Default: MAX(1,MAX(3*minmn+MAX(m,n),5*minmn))

`scipy.linalg.lapack.`**`dgesvd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `dgesvd`.

Parameters **a** : input rank-2 array('d') with bounds (m,n)

Returns **u** : rank-2 array('d') with bounds (u0,u1)

s : rank-1 array('d') with bounds (minmn)

vt : rank-2 array('d') with bounds (vt0,vt1)

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

compute_uv : input int, optional
Default: 1

full_matrices : input int, optional
Default: 1

lwork : input int, optional
Default: MAX(1,MAX(3*minmn+MAX(m,n),5*minmn))

`scipy.linalg.lapack.`**`cgesvd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cgesvd`.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **u** : rank-2 array('F') with bounds (u0,u1)

s : rank-1 array('f') with bounds (minmn)

vt : rank-2 array('F') with bounds (vt0,vt1)

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

compute_uv : input int, optional
Default: 1

full_matrices : input int, optional
Default: 1

lwork : input int, optional
Default: MAX(1,2*minmn+MAX(m,n))

`scipy.linalg.lapack.`**`zgesvd`**(*a*[, *compute_uv*, *full_matrices*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `zgesvd`.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **u** : rank-2 array('D') with bounds (u0,u1)

s : rank-1 array('d') with bounds (minmn)

vt : rank-2 array('D') with bounds (vt0,vt1)

info : int

Other Parameters

overwrite_a : input int, optional
  Default: 0

compute_uv : input int, optional
  Default: 1

full_matrices : input int, optional
  Default: 1

lwork : input int, optional
  Default: MAX(1,2*minmn+MAX(m,n))

`scipy.linalg.lapack.`**`sgesvd_lwork`**(*m*, *n*[, *compute_uv*, *full_matrices*]) **= <fortran object>**

Wrapper for `sgesvd_lwork`.

Parameters **m** : input int

n : input int

Returns **work** : float

info : int

Other Parameters

compute_uv : input int, optional
  Default: 1

full_matrices : input int, optional
  Default: 1

`scipy.linalg.lapack.`**`dgesvd_lwork`**(*m*, *n*[, *compute_uv*, *full_matrices*]) **= <fortran object>**

Wrapper for `dgesvd_lwork`.

Parameters **m** : input int

n : input int

Returns **work** : float

info : int

Other Parameters

compute_uv : input int, optional
  Default: 1

full_matrices : input int, optional
  Default: 1

`scipy.linalg.lapack.`**`cgesvd_lwork`**(*m*, *n*[, *compute_uv*, *full_matrices*]) **= <fortran object>**

Wrapper for `cgesvd_lwork`.

Parameters **m** : input int

n : input int

Returns **work** : complex

info : int

Other Parameters

compute_uv : input int, optional
  Default: 1

full_matrices : input int, optional
  Default: 1

scipy.linalg.lapack.**zgesvd_lwork**(*m*, *n*[, *compute_uv*, *full_matrices*]) = <fortran object>
Wrapper for `zgesvd_lwork`.

Parameters **m** : input int
n : input int

Returns **work** : complex
info : int

Other Parameters
compute_uv : input int, optional
Default: 1
full_matrices : input int, optional
Default: 1

scipy.linalg.lapack.**sgesv**(*a*, *b*[, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `sgesv`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns **lu** : rank-2 array('f') with bounds (n,n) and a storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**dgesv**(*a*, *b*[, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `dgesv`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **lu** : rank-2 array('d') with bounds (n,n) and a storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**cgesv**(*a*, *b*[, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `cgesv`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **lu** : rank-2 array('F') with bounds (n,n) and a storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**zgesv**(*a*, *b*[, *overwrite_a*, *overwrite_b*]) = **<fortran object>**
Wrapper for `zgesv`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **lu** : rank-2 array('D') with bounds (n,n) and a storage
piv : rank-1 array('i') with bounds (n)
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**sgesvx**(*a*, *b*[, *fact*, *trans*, *af*, *ipiv*, *equed*, *r*, *c*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**
Wrapper for `sgesvx`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns **as** : rank-2 array('f') with bounds (n,n) and a storage
lu : rank-2 array('f') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
equed : string(len=1)
rs : rank-1 array('f') with bounds (n) and r storage
cs : rank-1 array('f') with bounds (n) and c storage
bs : rank-2 array('f') with bounds (n,nrhs) and b storage
x : rank-2 array('f') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
trans : input string(len=1), optional
Default: 'N'
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('f') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
equed : input string(len=1), optional
Default: 'B'
r : input rank-1 array('f') with bounds (n)
c : input rank-1 array('f') with bounds (n)
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**dgesvx**(*a*, *b*[, *fact*, *trans*, *af*, *ipiv*, *equed*, *r*, *c*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**
Wrapper for `dgesvx`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **as** : rank-2 array('d') with bounds (n,n) and a storage
lu : rank-2 array('d') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
equed : string(len=1)
rs : rank-1 array('d') with bounds (n) and r storage
cs : rank-1 array('d') with bounds (n) and c storage
bs : rank-2 array('d') with bounds (n,nrhs) and b storage
x : rank-2 array('d') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
trans : input string(len=1), optional
Default: 'N'
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('d') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
equed : input string(len=1), optional
Default: 'B'
r : input rank-1 array('d') with bounds (n)
c : input rank-1 array('d') with bounds (n)
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cgesvx`**`(`*a*, *b*[, *fact*, *trans*, *af*, *ipiv*, *equed*, *r*, *c*, *overwrite_a*, *overwrite_b*]`)` **= <fortran object>**

Wrapper for `cgesvx`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **as** : rank-2 array('F') with bounds (n,n) and a storage
lu : rank-2 array('F') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
equed : string(len=1)
rs : rank-1 array('f') with bounds (n) and r storage
cs : rank-1 array('f') with bounds (n) and c storage
bs : rank-2 array('F') with bounds (n,nrhs) and b storage
x : rank-2 array('F') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
trans : input string(len=1), optional
Default: 'N'
overwrite_a : input int, optional
Default: 0

af : input rank-2 array('F') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
equed : input string(len=1), optional
Default: 'B'
r : input rank-1 array('f') with bounds (n)
c : input rank-1 array('f') with bounds (n)
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zgesvx`**(*a*, *b*[, *fact*, *trans*, *af*, *ipiv*, *equed*, *r*, *c*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `zgesvx`.

Parameters
a : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns
as : rank-2 array('D') with bounds (n,n) and a storage
lu : rank-2 array('D') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
equed : string(len=1)
rs : rank-1 array('d') with bounds (n) and r storage
cs : rank-1 array('d') with bounds (n) and c storage
bs : rank-2 array('D') with bounds (n,nrhs) and b storage
x : rank-2 array('D') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
trans : input string(len=1), optional
Default: 'N'
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('D') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
equed : input string(len=1), optional
Default: 'B'
r : input rank-1 array('d') with bounds (n)
c : input rank-1 array('d') with bounds (n)
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`sgecon`**(*a*, *anorm*[, *norm*]) = **<fortran object>**

Wrapper for `sgecon`.

Parameters
a : input rank-2 array('f') with bounds (n,n)
anorm : input float

Returns
rcond : float
info : int

Other Parameters
norm : input string(len=1), optional
Default: '1'

`scipy.linalg.lapack.`**`dgecon`**(*a*, *anorm*[, *norm*]) = **<fortran object>**

Wrapper for `dgecon`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
norm : input string(len=1), optional
Default: '1'

`scipy.linalg.lapack.`**`cgecon`**(*a*, *anorm*[, *norm*]) **= <fortran object>**
Wrapper for `cgecon`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
norm : input string(len=1), optional
Default: '1'

`scipy.linalg.lapack.`**`zgecon`**(*a*, *anorm*[, *norm*]) **= <fortran object>**
Wrapper for `zgecon`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
norm : input string(len=1), optional
Default: '1'

`scipy.linalg.lapack.`**`ssysv`**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**
Wrapper for `ssysv`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)
Returns **udut** : rank-2 array('f') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: n
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dsysv`**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**
Wrapper for `dsysv`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)
Returns **udut** : rank-2 array('d') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

lwork : input int, optional
Default: n

lower : input int, optional
Default: 0

scipy.linalg.lapack.**csysv**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for csysv.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **udut** : rank-2 array('F') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

lwork : input int, optional
Default: n

lower : input int, optional
Default: 0

scipy.linalg.lapack.**zsysv**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for zsysv.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **udut** : rank-2 array('D') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

lwork : input int, optional
Default: n

lower : input int, optional
Default: 0

scipy.linalg.lapack.**ssysv_lwork**(*n*[, *lower*]) = **<fortran object>**

Wrapper for ssysv_lwork.

Parameters **n** : input int

Returns **work** : float
info : int

Other Parameters

lower : input int, optional
Default: 0

scipy.linalg.lapack.**dsysv_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for dsysv_lwork.

Parameters **n** : input int
Returns **work** : float
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**csysv_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for csysv_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zsysv_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for zsysv_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**ssysvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for ssysvx.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)
Returns **a_s** : rank-2 array('f') with bounds (n,n) and a storage
udut : rank-2 array('f') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
b_s : rank-2 array('f') with bounds (n,nrhs) and b storage
x : rank-2 array('f') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('f') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n
factored : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**dsysvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `dsysvx`.

Parameters
a : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns
a_s : rank-2 array('d') with bounds (n,n) and a storage
udut : rank-2 array('d') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
b_s : rank-2 array('d') with bounds (n,nrhs) and b storage
x : rank-2 array('d') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('d') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n
factored : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**csysvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `csysvx`.

Parameters
a : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns
a_s : rank-2 array('F') with bounds (n,n) and a storage
udut : rank-2 array('F') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
b_s : rank-2 array('F') with bounds (n,nrhs) and b storage
x : rank-2 array('F') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('F') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n
factored : input int, optional
Default: 0
lower : input int, optional

Default: 0

scipy.linalg.lapack.**zsysvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for zsysvx.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **a_s** : rank-2 array('D') with bounds (n,n) and a storage
udut : rank-2 array('D') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
b_s : rank-2 array('D') with bounds (n,nrhs) and b storage
x : rank-2 array('D') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('D') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n
factored : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**ssysvx_lwork**(*n*[, *lower*]) = **<fortran object>**

Wrapper for ssysvx_lwork.

Parameters **n** : input int

Returns **work** : float
info : int

Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**dsysvx_lwork**(*n*[, *lower*]) = **<fortran object>**

Wrapper for dsysvx_lwork.

Parameters **n** : input int

Returns **work** : float
info : int

Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**csysvx_lwork**(*n*[, *lower*]) = **<fortran object>**

Wrapper for csysvx_lwork.

Parameters **n** : input int

Returns **work** : complex
info : int

Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zsysvx_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for `zsysvx_lwork`.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**chesv**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `chesv`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)
Returns **uduh** : rank-2 array('F') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: n
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zhesv**(*a*, *b*[, *lwork*, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `zhesv`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)
Returns **uduh** : rank-2 array('D') with bounds (n,n) and a storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: n
lower : input int, optional
Default: 0

scipy.linalg.lapack.**chesv_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for `chesv_lwork`.

Parameters **n** : input int
Returns **work** : complex
info : int

Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zhesv_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for `zhesv_lwork`.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**chesvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `chesvx`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)
Returns **uduh** : rank-2 array('F') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('F') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('F') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n
factored : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zhesvx**(*a*, *b*[, *af*, *ipiv*, *lwork*, *factored*, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `zhesvx`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)
Returns **uduh** : rank-2 array('D') with bounds (n,n) and af storage
ipiv : rank-1 array('i') with bounds (n)
x : rank-2 array('D') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0

af : input rank-2 array('D') with bounds (n,n)
ipiv : input rank-1 array('i') with bounds (n)
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n
factored : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**chesvx_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for chesvx_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zhesvx_lwork**(*n*[, *lower*]) = <fortran object>
Wrapper for zhesvx_lwork.

Parameters **n** : input int
Returns **work** : complex
info : int
Other Parameters
lower : input int, optional
Default: 0

scipy.linalg.lapack.**sgetrf**(*a*[, *overwrite_a*]) = <fortran object>
Wrapper for sgetrf.

Parameters **a** : input rank-2 array('f') with bounds (m,n)
Returns **lu** : rank-2 array('f') with bounds (m,n) and a storage
piv : rank-1 array('i') with bounds (MIN(m,n))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**dgetrf**(*a*[, *overwrite_a*]) = <fortran object>
Wrapper for dgetrf.

Parameters **a** : input rank-2 array('d') with bounds (m,n)
Returns **lu** : rank-2 array('d') with bounds (m,n) and a storage
piv : rank-1 array('i') with bounds (MIN(m,n))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**cgetrf**(*a*[, *overwrite_a*]) = <fortran object>
Wrapper for cgetrf.

Parameters **a** : input rank-2 array('F') with bounds (m,n)

Returns **lu** : rank-2 array('F') with bounds (m,n) and a storage
piv : rank-1 array('i') with bounds (MIN(m,n))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**zgetrf**(*a*[, *overwrite_a*]) = <fortran object>
Wrapper for zgetrf.

Parameters **a** : input rank-2 array('D') with bounds (m,n)

Returns **lu** : rank-2 array('D') with bounds (m,n) and a storage
piv : rank-1 array('i') with bounds (MIN(m,n))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0

scipy.linalg.lapack.**sgetri**(*lu*, *piv*[, *lwork*, *overwrite_lu*]) = <fortran object>
Wrapper for sgetri.

Parameters **lu** : input rank-2 array('f') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)

Returns **inv_a** : rank-2 array('f') with bounds (n,n) and lu storage
info : int

Other Parameters
overwrite_lu : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

scipy.linalg.lapack.**dgetri**(*lu*, *piv*[, *lwork*, *overwrite_lu*]) = <fortran object>
Wrapper for dgetri.

Parameters **lu** : input rank-2 array('d') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)

Returns **inv_a** : rank-2 array('d') with bounds (n,n) and lu storage
info : int

Other Parameters
overwrite_lu : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

scipy.linalg.lapack.**cgetri**(*lu*, *piv*[, *lwork*, *overwrite_lu*]) = <fortran object>
Wrapper for cgetri.

Parameters **lu** : input rank-2 array('F') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)

Returns **inv_a** : rank-2 array('F') with bounds (n,n) and lu storage
info : int

Other Parameters
overwrite_lu : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

`scipy.linalg.lapack.`**`zgetri`**(*lu*, *piv*[, *lwork*, *overwrite_lu*]) = **<fortran object>**
Wrapper for `zgetri`.

Parameters **lu** : input rank-2 array('D') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)
Returns **inv_a** : rank-2 array('D') with bounds (n,n) and lu storage
info : int
Other Parameters
overwrite_lu : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

`scipy.linalg.lapack.`**`sgetri_lwork`**(*n*) = **<fortran object>**
Wrapper for `sgetri_lwork`.

Parameters **n** : input int
Returns **work** : float
info : int

`scipy.linalg.lapack.`**`dgetri_lwork`**(*n*) = **<fortran object>**
Wrapper for `dgetri_lwork`.

Parameters **n** : input int
Returns **work** : float
info : int

`scipy.linalg.lapack.`**`cgetri_lwork`**(*n*) = **<fortran object>**
Wrapper for `cgetri_lwork`.

Parameters **n** : input int
Returns **work** : complex
info : int

`scipy.linalg.lapack.`**`zgetri_lwork`**(*n*) = **<fortran object>**
Wrapper for `zgetri_lwork`.

Parameters **n** : input int
Returns **work** : complex
info : int

`scipy.linalg.lapack.`**`sgetrs`**(*lu*, *piv*, *b*[, *trans*, *overwrite_b*]) = **<fortran object>**
Wrapper for `sgetrs`.

Parameters **lu** : input rank-2 array('f') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)
b : input rank-2 array('f') with bounds (n,nrhs)
Returns **x** : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dgetrs`**(*lu*, *piv*, *b*[, *trans*, *overwrite_b*]) = **<fortran object>**
Wrapper for `dgetrs`.

Parameters **lu** : input rank-2 array('d') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **x** : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0

scipy.linalg.lapack.**cgetrs**(*lu*, *piv*, *b*[, *trans*, *overwrite_b*]) **= <fortran object>**
Wrapper for `cgetrs`.

Parameters **lu** : input rank-2 array('F') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **x** : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0

scipy.linalg.lapack.**zgetrs**(*lu*, *piv*, *b*[, *trans*, *overwrite_b*]) **= <fortran object>**
Wrapper for `zgetrs`.

Parameters **lu** : input rank-2 array('D') with bounds (n,n)
piv : input rank-1 array('i') with bounds (n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **x** : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
trans : input int, optional
Default: 0

scipy.linalg.lapack.**sgges**(*sselect*, *a*, *b*[, *jobvsl*, *jobvsr*, *sort_t*, *ldvsl*, *ldvsr*, *lwork*, *sselect_extra_args*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**
Wrapper for `sgges`.

Parameters **sselect** : call-back function
a : input rank-2 array('f') with bounds (lda,n)
b : input rank-2 array('f') with bounds (ldb,n)

Returns **a** : rank-2 array('f') with bounds (lda,n)
b : rank-2 array('f') with bounds (ldb,n)
sdim : int
alphar : rank-1 array('f') with bounds (n)
alphai : rank-1 array('f') with bounds (n)
beta : rank-1 array('f') with bounds (n)
vsl : rank-2 array('f') with bounds (ldvsl,n)
vsr : rank-2 array('f') with bounds (ldvsr,n)
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int

Other Parameters

jobvsl : input int, optional
    Default: 1
jobvsr : input int, optional
    Default: 1
sort_t : input int, optional
    Default: 0
sselect_extra_args : input tuple, optional
    Default: ()
overwrite_a : input int, optional
    Default: 0
overwrite_b : input int, optional
    Default: 0
ldvsl : input int, optional
    Default: ((jobvsl==1)?n:1)
ldvsr : input int, optional
    Default: ((jobvsr==1)?n:1)
lwork : input int, optional
    Default: 8*n+16

Notes

Call-back functions:

```
def sselect(alphar,alphai,beta): return sselect
Required arguments:
  alphar : input float
  alphai : input float
  beta : input float
Return objects:
  sselect : int
```

scipy.linalg.lapack.**dgges**(*dselect, a, b*[, *jobvsl, jobvsr, sort_t, ldvsl, ldvsr, lwork, dselect_extra_args, overwrite_a, overwrite_b*]) = **<fortran object>**

Wrapper for `dgges`.

Parameters

dselect : call-back function
a : input rank-2 array('d') with bounds (lda,n)
b : input rank-2 array('d') with bounds (ldb,n)

Returns

a : rank-2 array('d') with bounds (lda,n)
b : rank-2 array('d') with bounds (ldb,n)
sdim : int
alphar : rank-1 array('d') with bounds (n)
alphai : rank-1 array('d') with bounds (n)
beta : rank-1 array('d') with bounds (n)
vsl : rank-2 array('d') with bounds (ldvsl,n)
vsr : rank-2 array('d') with bounds (ldvsr,n)
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int

Other Parameters

jobvsl : input int, optional
    Default: 1
jobvsr : input int, optional
    Default: 1
sort_t : input int, optional
    Default: 0

dselect_extra_args : input tuple, optional
    Default: ()
overwrite_a : input int, optional
    Default: 0
overwrite_b : input int, optional
    Default: 0
ldvsl : input int, optional
    Default: ((jobvsl==1)?n:1)
ldvsr : input int, optional
    Default: ((jobvsr==1)?n:1)
lwork : input int, optional
    Default: 8*n+16

Notes

Call-back functions:

```
def dselect(alphar,alphai,beta): return dselect
Required arguments:
  alphar : input float
  alphai : input float
  beta : input float
Return objects:
  dselect : int
```

scipy.linalg.lapack.**cgges**(*cselect*, *a*, *b*[, *jobvsl*, *jobvsr*, *sort_t*, *ldvsl*, *ldvsr*, *lwork*, *cselect_extra_args*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `cgges`.

Parameters **cselect** : call-back function
a : input rank-2 array('F') with bounds (lda,n)
b : input rank-2 array('F') with bounds (ldb,n)

Returns **a** : rank-2 array('F') with bounds (lda,n)
b : rank-2 array('F') with bounds (ldb,n)
sdim : int
alpha : rank-1 array('F') with bounds (n)
beta : rank-1 array('F') with bounds (n)
vsl : rank-2 array('F') with bounds (ldvsl,n)
vsr : rank-2 array('F') with bounds (ldvsr,n)
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int

Other Parameters
jobvsl : input int, optional
    Default: 1
jobvsr : input int, optional
    Default: 1
sort_t : input int, optional
    Default: 0
cselect_extra_args : input tuple, optional
    Default: ()
overwrite_a : input int, optional
    Default: 0
overwrite_b : input int, optional
    Default: 0
ldvsl : input int, optional
    Default: ((jobvsl==1)?n:1)

ldvsr : input int, optional
Default: ((jobvsr==1)?n:1)
lwork : input int, optional
Default: 2*n

Notes

Call-back functions:

```
def cselect(alpha,beta): return cselect
Required arguments:
  alpha : input complex
  beta : input complex
Return objects:
  cselect : int
```

`scipy.linalg.lapack.`**`zgges`**(*zselect*, *a*, *b*[, *jobvsl*, *jobvsr*, *sort_t*, *ldvsl*, *ldvsr*, *lwork*, *zselect_extra_args*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zgges`.

Parameters
zselect : call-back function
a : input rank-2 array('D') with bounds (lda,n)
b : input rank-2 array('D') with bounds (ldb,n)

Returns
a : rank-2 array('D') with bounds (lda,n)
b : rank-2 array('D') with bounds (ldb,n)
sdim : int
alpha : rank-1 array('D') with bounds (n)
beta : rank-1 array('D') with bounds (n)
vsl : rank-2 array('D') with bounds (ldvsl,n)
vsr : rank-2 array('D') with bounds (ldvsr,n)
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
jobvsl : input int, optional
Default: 1
jobvsr : input int, optional
Default: 1
sort_t : input int, optional
Default: 0
zselect_extra_args : input tuple, optional
Default: ()
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
ldvsl : input int, optional
Default: ((jobvsl==1)?n:1)
ldvsr : input int, optional
Default: ((jobvsr==1)?n:1)
lwork : input int, optional
Default: 2*n

Notes

Call-back functions:

```
def zselect(alpha,beta): return zselect
Required arguments:
  alpha : input complex
  beta : input complex
Return objects:
  zselect : int
```

`scipy.linalg.lapack.`**`sggev`**(*a*, *b*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `sggev`.

Parameters
: **a** : input rank-2 array('f') with bounds (n,n)
: **b** : input rank-2 array('f') with bounds (n,n)

Returns
: **alphar** : rank-1 array('f') with bounds (n)
: **alphai** : rank-1 array('f') with bounds (n)
: **beta** : rank-1 array('f') with bounds (n)
: **vl** : rank-2 array('f') with bounds (ldvl,n)
: **vr** : rank-2 array('f') with bounds (ldvr,n)
: **work** : rank-1 array('f') with bounds (MAX(lwork,1))
: **info** : int

Other Parameters
: **compute_vl** : input int, optional
  Default: 1
: **compute_vr** : input int, optional
  Default: 1
: **overwrite_a** : input int, optional
  Default: 0
: **overwrite_b** : input int, optional
  Default: 0
: **lwork** : input int, optional
  Default: 8*n

`scipy.linalg.lapack.`**`dggev`**(*a*, *b*[, *compute_vl*, *compute_vr*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `dggev`.

Parameters
: **a** : input rank-2 array('d') with bounds (n,n)
: **b** : input rank-2 array('d') with bounds (n,n)

Returns
: **alphar** : rank-1 array('d') with bounds (n)
: **alphai** : rank-1 array('d') with bounds (n)
: **beta** : rank-1 array('d') with bounds (n)
: **vl** : rank-2 array('d') with bounds (ldvl,n)
: **vr** : rank-2 array('d') with bounds (ldvr,n)
: **work** : rank-1 array('d') with bounds (MAX(lwork,1))
: **info** : int

Other Parameters
: **compute_vl** : input int, optional
  Default: 1
: **compute_vr** : input int, optional
  Default: 1
: **overwrite_a** : input int, optional
  Default: 0
: **overwrite_b** : input int, optional
  Default: 0
: **lwork** : input int, optional
  Default: 8*n

`scipy.linalg.lapack.cggev(a, b[, compute_vl, compute_vr, lwork, overwrite_a, overwrite_b]) = <fortran object>`

Wrapper for `cggev`.

Parameters
a : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,n)

Returns
alpha : rank-1 array('F') with bounds (n)
beta : rank-1 array('F') with bounds (n)
vl : rank-2 array('F') with bounds (ldvl,n)
vr : rank-2 array('F') with bounds (ldvr,n)
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int

Other Parameters
compute_vl : input int, optional
Default: 1
compute_vr : input int, optional
Default: 1
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n

`scipy.linalg.lapack.zggev(a, b[, compute_vl, compute_vr, lwork, overwrite_a, overwrite_b]) = <fortran object>`

Wrapper for `zggev`.

Parameters
a : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,n)

Returns
alpha : rank-1 array('D') with bounds (n)
beta : rank-1 array('D') with bounds (n)
vl : rank-2 array('D') with bounds (ldvl,n)
vr : rank-2 array('D') with bounds (ldvr,n)
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
compute_vl : input int, optional
Default: 1
compute_vr : input int, optional
Default: 1
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n

`scipy.linalg.lapack.chbevd(ab[, compute_v, lower, ldab, lrwork, liwork, overwrite_ab]) = <fortran object>`

Wrapper for `chbevd`.

Parameters
ab : input rank-2 array('F') with bounds (ldab,n)

Returns
w : rank-1 array('f') with bounds (n)
z : rank-2 array('F') with bounds (ldz,ldz)
info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
lrwork : input int, optional
Default: (compute_v?1+5*n+2*n*n:n)
liwork : input int, optional
Default: (compute_v?3+5*n:1)

scipy.linalg.lapack.**zhbevd**(*ab*[, *compute_v*, *lower*, *ldab*, *lrwork*, *liwork*, *overwrite_ab*]) **= <fortran object>**

Wrapper for zhbevd.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,n)

Returns **w** : rank-1 array('d') with bounds (n)
z : rank-2 array('D') with bounds (ldz,ldz)
info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
lrwork : input int, optional
Default: (compute_v?1+5*n+2*n*n:n)
liwork : input int, optional
Default: (compute_v?3+5*n:1)

scipy.linalg.lapack.**chbevx**(*ab*, *vl*, *vu*, *il*, *iu*[, *ldab*, *compute_v*, *range*, *lower*, *abstol*, *mmax*, *overwrite_ab*]) **= <fortran object>**

Wrapper for chbevx.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,n)
vl : input float
vu : input float
il : input int
iu : input int

Returns **w** : rank-1 array('f') with bounds (n)
z : rank-2 array('F') with bounds (ldz,mmax)
m : int
ifail : rank-1 array('i') with bounds ((compute_v?n:1))
info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1
ldab : input int, optional
Default: shape(ab,0)
compute_v : input int, optional

Default: 1
range : input int, optional
Default: 0
lower : input int, optional
Default: 0
abstol : input float, optional
Default: 0.0
mmax : input int, optional
Default: (compute_v?(range==2?(iu-il+1):n):1)

`scipy.linalg.lapack.`**`zhbevx`**(*ab*, *vl*, *vu*, *il*, *iu*[, *ldab*, *compute_v*, *range*, *lower*, *abstol*, *mmax*, *overwrite_ab*]) **= <fortran object>**

Wrapper for `zhbevx`.

Parameters
ab : input rank-2 array('D') with bounds (ldab,n)
vl : input float
vu : input float
il : input int
iu : input int

Returns
w : rank-1 array('d') with bounds (n)
z : rank-2 array('D') with bounds (ldz,mmax)
m : int
ifail : rank-1 array('i') with bounds ((compute_v?n:1))
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 1
ldab : input int, optional
Default: shape(ab,0)
compute_v : input int, optional
Default: 1
range : input int, optional
Default: 0
lower : input int, optional
Default: 0
abstol : input float, optional
Default: 0.0
mmax : input int, optional
Default: (compute_v?(range==2?(iu-il+1):n):1)

`scipy.linalg.lapack.`**`cheev`**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cheev`.

Parameters
a : input rank-2 array('F') with bounds (n,n)

Returns
w : rank-1 array('f') with bounds (n)
v : rank-2 array('F') with bounds (n,n) and a storage
info : int

Other Parameters
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n-1

`scipy.linalg.lapack.`**`zheev`**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zheev`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
Returns **w** : rank-1 array('d') with bounds (n)
v : rank-2 array('D') with bounds (n,n) and a storage
info : int
Other Parameters
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n-1

`scipy.linalg.lapack.`**`cheevd`**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `cheevd`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
Returns **w** : rank-1 array('f') with bounds (n)
v : rank-2 array('F') with bounds (n,n) and a storage
info : int
Other Parameters
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: (compute_v?2*n+n*n:n+1)

`scipy.linalg.lapack.`**`zheevd`**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zheevd`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
Returns **w** : rank-1 array('d') with bounds (n)
v : rank-2 array('D') with bounds (n,n) and a storage
info : int
Other Parameters
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: (compute_v?2*n+n*n:n+1)

`scipy.linalg.lapack.`**`cheevr`**(*a*[, *jobz*, *range*, *uplo*, *il*, *iu*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `cheevr`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)

Returns **w** : rank-1 array('f') with bounds (n)
z : rank-2 array('F') with bounds (n,m)
info : int

Other Parameters
jobz : input string(len=1), optional
Default: 'V'
range : input string(len=1), optional
Default: 'A'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
il : input int, optional
Default: 1
iu : input int, optional
Default: n
lwork : input int, optional
Default: 18*n

`scipy.linalg.lapack.`**`zheevr`**(*a*[, *jobz*, *range*, *uplo*, *il*, *iu*, *lwork*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zheevr`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
Returns **w** : rank-1 array('d') with bounds (n)
z : rank-2 array('D') with bounds (n,m)
info : int

Other Parameters
jobz : input string(len=1), optional
Default: 'V'
range : input string(len=1), optional
Default: 'A'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
il : input int, optional
Default: 1
iu : input int, optional
Default: n
lwork : input int, optional
Default: 18*n

`scipy.linalg.lapack.`**`chegv`**(*a*, *b*[, *itype*, *jobz*, *uplo*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**
Wrapper for `chegv`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,n)
Returns **a** : rank-2 array('F') with bounds (n,n)
w : rank-1 array('f') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional

Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zhegv`**(*a*, *b*[, *itype*, *jobz*, *uplo*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zhegv`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,n)

Returns **a** : rank-2 array('D') with bounds (n,n)
w : rank-1 array('d') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`chegvd`**(*a*, *b*[, *itype*, *jobz*, *uplo*, *lwork*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `chegvd`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,n)

Returns **a** : rank-2 array('F') with bounds (n,n)
w : rank-1 array('f') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n+n*n

`scipy.linalg.lapack.`**`zhegvd`**(*a*, *b*[, *itype*, *jobz*, *uplo*, *lwork*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zhegvd`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,n)

Returns
a : rank-2 array('D') with bounds (n,n)
w : rank-1 array('d') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 2*n+n*n

scipy.linalg.lapack.**chegvx**(*a*, *b*, *iu*[, *itype*, *jobz*, *uplo*, *il*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `chegvx`.

Parameters
a : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,n)
iu : input int

Returns
w : rank-1 array('f') with bounds (n)
z : rank-2 array('F') with bounds (n,m)
ifail : rank-1 array('i') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
il : input int, optional
Default: 1
lwork : input int, optional
Default: 18*n-1

scipy.linalg.lapack.**zhegvx**(*a*, *b*, *iu*[, *itype*, *jobz*, *uplo*, *il*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `zhegvx`.

Parameters
a : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,n)
iu : input int

Returns
w : rank-1 array('d') with bounds (n)
z : rank-2 array('D') with bounds (n,m)
ifail : rank-1 array('i') with bounds (n)
info : int

Other Parameters

itype : input int, optional
Default: 1

jobz : input string(len=1), optional
Default: 'V'

uplo : input string(len=1), optional
Default: 'L'

overwrite_a : input int, optional
Default: 0

overwrite_b : input int, optional
Default: 0

il : input int, optional
Default: 1

lwork : input int, optional
Default: 18*n-1

`scipy.linalg.lapack.`**`slarf`**(*v*, *tau*, *c*, *work*[, *side*, *incv*, *overwrite_c*]) **= <fortran object>**

Wrapper for `slarf`.

Parameters
v : input rank-1 array('f') with bounds (*)
tau : input float
c : input rank-2 array('f') with bounds (m,n)
work : input rank-1 array('f') with bounds (*)

Returns
c : rank-2 array('f') with bounds (m,n)

Other Parameters

side : input string(len=1), optional
Default: 'L'

incv : input int, optional
Default: 1

overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dlarf`**(*v*, *tau*, *c*, *work*[, *side*, *incv*, *overwrite_c*]) **= <fortran object>**

Wrapper for `dlarf`.

Parameters
v : input rank-1 array('d') with bounds (*)
tau : input float
c : input rank-2 array('d') with bounds (m,n)
work : input rank-1 array('d') with bounds (*)

Returns
c : rank-2 array('d') with bounds (m,n)

Other Parameters

side : input string(len=1), optional
Default: 'L'

incv : input int, optional
Default: 1

overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`clarf`**(*v*, *tau*, *c*, *work*[, *side*, *incv*, *overwrite_c*]) **= <fortran object>**

Wrapper for `clarf`.

Parameters
v : input rank-1 array('F') with bounds (*)
tau : input complex
c : input rank-2 array('F') with bounds (m,n)
work : input rank-1 array('F') with bounds (*)

Returns
c : rank-2 array('F') with bounds (m,n)

Other Parameters

side : input string(len=1), optional
Default: 'L'
incv : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

scipy.linalg.lapack.**zlarf**(*v*, *tau*, *c*, *work*[, *side*, *incv*, *overwrite_c*]) = **<fortran object>**
Wrapper for zlarf.

Parameters **v** : input rank-1 array('D') with bounds (*)
tau : input complex
c : input rank-2 array('D') with bounds (m,n)
work : input rank-1 array('D') with bounds (*)
Returns **c** : rank-2 array('D') with bounds (m,n)
Other Parameters
side : input string(len=1), optional
Default: 'L'
incv : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

scipy.linalg.lapack.**slarfg**(*n*, *alpha*, *x*[, *incx*, *overwrite_x*]) = **<fortran object>**
Wrapper for slarfg.

Parameters **n** : input int
alpha : input float
x : input rank-1 array('f') with bounds (*)
Returns **alpha** : float
x : rank-1 array('f') with bounds (*)
tau : float
Other Parameters
overwrite_x : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.lapack.**dlarfg**(*n*, *alpha*, *x*[, *incx*, *overwrite_x*]) = **<fortran object>**
Wrapper for dlarfg.

Parameters **n** : input int
alpha : input float
x : input rank-1 array('d') with bounds (*)
Returns **alpha** : float
x : rank-1 array('d') with bounds (*)
tau : float
Other Parameters
overwrite_x : input int, optional
Default: 0
incx : input int, optional
Default: 1

scipy.linalg.lapack.**clarfg**(*n*, *alpha*, *x*[, *incx*, *overwrite_x*]) = **<fortran object>**
Wrapper for clarfg.

Parameters **n** : input int
alpha : input complex
x : input rank-1 array('F') with bounds (*)

Returns **alpha** : complex
x : rank-1 array('F') with bounds (*)
tau : complex

Other Parameters
overwrite_x : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.lapack.zlarfg`(*n*, *alpha*, *x*[, *incx*, *overwrite_x*]) = **<fortran object>**
Wrapper for `zlarfg`.

Parameters **n** : input int
alpha : input complex
x : input rank-1 array('D') with bounds (*)

Returns **alpha** : complex
x : rank-1 array('D') with bounds (*)
tau : complex

Other Parameters
overwrite_x : input int, optional
Default: 0
incx : input int, optional
Default: 1

`scipy.linalg.lapack.slartg`(*f*, *g*) = **<fortran object>**
Wrapper for `slartg`.

Parameters **f** : input float
g : input float

Returns **cs** : float
sn : float
r : float

`scipy.linalg.lapack.dlartg`(*f*, *g*) = **<fortran object>**
Wrapper for `dlartg`.

Parameters **f** : input float
g : input float

Returns **cs** : float
sn : float
r : float

`scipy.linalg.lapack.clartg`(*f*, *g*) = **<fortran object>**
Wrapper for `clartg`.

Parameters **f** : input complex
g : input complex

Returns **cs** : float
sn : complex
r : complex

`scipy.linalg.lapack.zlartg`(*f*, *g*) = **<fortran object>**
Wrapper for `zlartg`.

Parameters **f** : input complex
g : input complex

Returns **cs** : float
sn : complex
r : complex

`scipy.linalg.lapack.`**`slasd4`**(*i*, *d*, *z*[, *rho*]) **= <fortran object>**
Wrapper for `slasd4`.

Parameters **i** : input int
d : input rank-1 array('f') with bounds (n)
z : input rank-1 array('f') with bounds (n)
Returns **delta** : rank-1 array('f') with bounds (n)
sigma : float
work : rank-1 array('f') with bounds (n)
info : int
Other Parameters
rho : input float, optional
Default: 1.0

`scipy.linalg.lapack.`**`dlasd4`**(*i*, *d*, *z*[, *rho*]) **= <fortran object>**
Wrapper for `dlasd4`.

Parameters **i** : input int
d : input rank-1 array('d') with bounds (n)
z : input rank-1 array('d') with bounds (n)
Returns **delta** : rank-1 array('d') with bounds (n)
sigma : float
work : rank-1 array('d') with bounds (n)
info : int
Other Parameters
rho : input float, optional
Default: 1.0

`scipy.linalg.lapack.`**`slaswp`**(*a*, *piv*[, *k1*, *k2*, *off*, *inc*, *overwrite_a*]) **= <fortran object>**
Wrapper for `slaswp`.

Parameters **a** : input rank-2 array('f') with bounds (nrows,n)
piv : input rank-1 array('i') with bounds (*)
Returns **a** : rank-2 array('f') with bounds (nrows,n)
Other Parameters
overwrite_a : input int, optional
Default: 0
k1 : input int, optional
Default: 0
k2 : input int, optional
Default: len(piv)-1
off : input int, optional
Default: 0
inc : input int, optional
Default: 1

`scipy.linalg.lapack.`**`dlaswp`**(*a*, *piv*[, *k1*, *k2*, *off*, *inc*, *overwrite_a*]) **= <fortran object>**
Wrapper for `dlaswp`.

Parameters **a** : input rank-2 array('d') with bounds (nrows,n)
piv : input rank-1 array('i') with bounds (*)
Returns **a** : rank-2 array('d') with bounds (nrows,n)
Other Parameters
overwrite_a : input int, optional

Default: 0
k1 : input int, optional
Default: 0
k2 : input int, optional
Default: len(piv)-1
off : input int, optional
Default: 0
inc : input int, optional
Default: 1

scipy.linalg.lapack.**claswp**(*a*, *piv*[, *k1*, *k2*, *off*, *inc*, *overwrite_a*]) = <fortran object>
Wrapper for `claswp`.

Parameters **a** : input rank-2 array('F') with bounds (nrows,n)
piv : input rank-1 array('i') with bounds (*)
Returns **a** : rank-2 array('F') with bounds (nrows,n)
Other Parameters
overwrite_a : input int, optional
Default: 0
k1 : input int, optional
Default: 0
k2 : input int, optional
Default: len(piv)-1
off : input int, optional
Default: 0
inc : input int, optional
Default: 1

scipy.linalg.lapack.**zlaswp**(*a*, *piv*[, *k1*, *k2*, *off*, *inc*, *overwrite_a*]) = <fortran object>
Wrapper for `zlaswp`.

Parameters **a** : input rank-2 array('D') with bounds (nrows,n)
piv : input rank-1 array('i') with bounds (*)
Returns **a** : rank-2 array('D') with bounds (nrows,n)
Other Parameters
overwrite_a : input int, optional
Default: 0
k1 : input int, optional
Default: 0
k2 : input int, optional
Default: len(piv)-1
off : input int, optional
Default: 0
inc : input int, optional
Default: 1

scipy.linalg.lapack.**slauum**(*c*[, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for `slauum`.

Parameters **c** : input rank-2 array('f') with bounds (n,n)
Returns **a** : rank-2 array('f') with bounds (n,n) and c storage
info : int
Other Parameters
overwrite_c : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**dlauum**(*c*[, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for dlauum.

Parameters **c** : input rank-2 array('d') with bounds (n,n)
Returns **a** : rank-2 array('d') with bounds (n,n) and c storage
info : int
Other Parameters
overwrite_c : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**clauum**(*c*[, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for clauum.

Parameters **c** : input rank-2 array('F') with bounds (n,n)
Returns **a** : rank-2 array('F') with bounds (n,n) and c storage
info : int
Other Parameters
overwrite_c : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**zlauum**(*c*[, *lower*, *overwrite_c*]) = <fortran object>
Wrapper for zlauum.

Parameters **c** : input rank-2 array('D') with bounds (n,n)
Returns **a** : rank-2 array('D') with bounds (n,n) and c storage
info : int
Other Parameters
overwrite_c : input int, optional
Default: 0
lower : input int, optional
Default: 0

scipy.linalg.lapack.**spbsv**(*ab*, *b*[, *lower*, *ldab*, *overwrite_ab*, *overwrite_b*]) = <fortran object>
Wrapper for spbsv.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)
b : input rank-2 array('f') with bounds (ldb,nrhs)
Returns **c** : rank-2 array('f') with bounds (ldab,n) and ab storage
x : rank-2 array('f') with bounds (ldb,nrhs) and b storage
info : int
Other Parameters
lower : input int, optional
Default: 0
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**dpbsv**(*ab*, *b*[, *lower*, *ldab*, *overwrite_ab*, *overwrite_b*]) = <fortran object>
Wrapper for dpbsv.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)
b : input rank-2 array('d') with bounds (ldb,nrhs)

Returns **c** : rank-2 array('d') with bounds (ldab,n) and ab storage
x : rank-2 array('d') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters
lower : input int, optional
Default: 0
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**cpbsv**(*ab*, *b*[, *lower*, *ldab*, *overwrite_ab*, *overwrite_b*]) **= <fortran object>**
Wrapper for cpbsv.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,n)
b : input rank-2 array('F') with bounds (ldb,nrhs)

Returns **c** : rank-2 array('F') with bounds (ldab,n) and ab storage
x : rank-2 array('F') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters
lower : input int, optional
Default: 0
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**zpbsv**(*ab*, *b*[, *lower*, *ldab*, *overwrite_ab*, *overwrite_b*]) **= <fortran object>**
Wrapper for zpbsv.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,n)
b : input rank-2 array('D') with bounds (ldb,nrhs)

Returns **c** : rank-2 array('D') with bounds (ldab,n) and ab storage
x : rank-2 array('D') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters
lower : input int, optional
Default: 0
overwrite_ab : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**spbtrf**(*ab*[, *lower*, *ldab*, *overwrite_ab*]) **= <fortran object>**
Wrapper for spbtrf.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

Returns **c** : rank-2 array('f') with bounds (ldab,n) and ab storage
info : int

Other Parameters

lower : input int, optional
Default: 0

overwrite_ab : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.`**`dpbtrf`**(*ab*[, *lower*, *ldab*, *overwrite_ab*]) = <fortran object>

Wrapper for `dpbtrf`.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)

Returns **c** : rank-2 array('d') with bounds (ldab,n) and ab storage

info : int

Other Parameters

lower : input int, optional
Default: 0

overwrite_ab : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.`**`cpbtrf`**(*ab*[, *lower*, *ldab*, *overwrite_ab*]) = <fortran object>

Wrapper for `cpbtrf`.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,n)

Returns **c** : rank-2 array('F') with bounds (ldab,n) and ab storage

info : int

Other Parameters

lower : input int, optional
Default: 0

overwrite_ab : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.`**`zpbtrf`**(*ab*[, *lower*, *ldab*, *overwrite_ab*]) = <fortran object>

Wrapper for `zpbtrf`.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,n)

Returns **c** : rank-2 array('D') with bounds (ldab,n) and ab storage

info : int

Other Parameters

lower : input int, optional
Default: 0

overwrite_ab : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.`**`spbtrs`**(*ab*, *b*[, *lower*, *ldab*, *overwrite_b*]) = <fortran object>

Wrapper for `spbtrs`.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

b : input rank-2 array('f') with bounds (ldb,nrhs)

Returns **x** : rank-2 array('f') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dpbtrs`**(*ab*, *b*[, *lower*, *ldab*, *overwrite_b*]) = <fortran object>

Wrapper for `dpbtrs`.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)
b : input rank-2 array('d') with bounds (ldb,nrhs)

Returns **x** : rank-2 array('d') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cpbtrs`**(*ab*, *b*[, *lower*, *ldab*, *overwrite_b*]) = <fortran object>

Wrapper for `cpbtrs`.

Parameters **ab** : input rank-2 array('F') with bounds (ldab,n)
b : input rank-2 array('F') with bounds (ldb,nrhs)

Returns **x** : rank-2 array('F') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zpbtrs`**(*ab*, *b*[, *lower*, *ldab*, *overwrite_b*]) = <fortran object>

Wrapper for `zpbtrs`.

Parameters **ab** : input rank-2 array('D') with bounds (ldab,n)
b : input rank-2 array('D') with bounds (ldb,nrhs)

Returns **x** : rank-2 array('D') with bounds (ldb,nrhs) and b storage
info : int

Other Parameters

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`sposv`**(*a*, *b*[, *lower*, *overwrite_a*, *overwrite_b*]) = <fortran object>

Wrapper for `sposv`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns **c** : rank-2 array('f') with bounds (n,n) and a storage
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dposv`**(*a*, *b*[, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dposv`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **c** : rank-2 array('d') with bounds (n,n) and a storage
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cposv`**(*a*, *b*[, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `cposv`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **c** : rank-2 array('F') with bounds (n,n) and a storage
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zposv`**(*a*, *b*[, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zposv`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **c** : rank-2 array('D') with bounds (n,n) and a storage
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lower : input int, optional

Default: 0

`scipy.linalg.lapack.sposvx`(*a*, *b*[, *fact*, *af*, *equed*, *s*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `sposvx`.

Parameters
- **a** : input rank-2 array('f') with bounds (n,n)
- **b** : input rank-2 array('f') with bounds (n,nrhs)

Returns
- **a_s** : rank-2 array('f') with bounds (n,n) and a storage
- **lu** : rank-2 array('f') with bounds (n,n) and af storage
- **equed** : string(len=1)
- **s** : rank-1 array('f') with bounds (n)
- **b_s** : rank-2 array('f') with bounds (n,nrhs) and b storage
- **x** : rank-2 array('f') with bounds (n,nrhs)
- **rcond** : float
- **ferr** : rank-1 array('f') with bounds (nrhs)
- **berr** : rank-1 array('f') with bounds (nrhs)
- **info** : int

Other Parameters
- **fact** : input string(len=1), optional
  Default: 'E'
- **overwrite_a** : input int, optional
  Default: 0
- **af** : input rank-2 array('f') with bounds (n,n)
- **equed** : input string(len=1), optional
  Default: 'Y'
- **s** : input rank-1 array('f') with bounds (n)
- **overwrite_b** : input int, optional
  Default: 0
- **lower** : input int, optional
  Default: 0

`scipy.linalg.lapack.dposvx`(*a*, *b*[, *fact*, *af*, *equed*, *s*, *lower*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `dposvx`.

Parameters
- **a** : input rank-2 array('d') with bounds (n,n)
- **b** : input rank-2 array('d') with bounds (n,nrhs)

Returns
- **a_s** : rank-2 array('d') with bounds (n,n) and a storage
- **lu** : rank-2 array('d') with bounds (n,n) and af storage
- **equed** : string(len=1)
- **s** : rank-1 array('d') with bounds (n)
- **b_s** : rank-2 array('d') with bounds (n,nrhs) and b storage
- **x** : rank-2 array('d') with bounds (n,nrhs)
- **rcond** : float
- **ferr** : rank-1 array('d') with bounds (nrhs)
- **berr** : rank-1 array('d') with bounds (nrhs)
- **info** : int

Other Parameters
- **fact** : input string(len=1), optional
  Default: 'E'
- **overwrite_a** : input int, optional
  Default: 0
- **af** : input rank-2 array('d') with bounds (n,n)
- **equed** : input string(len=1), optional
  Default: 'Y'

s : input rank-1 array('d') with bounds (n)
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cposvx`**(*a*, *b*[, *fact*, *af*, *equed*, *s*, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `cposvx`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **a_s** : rank-2 array('F') with bounds (n,n) and a storage
lu : rank-2 array('F') with bounds (n,n) and af storage
equed : string(len=1)
s : rank-1 array('f') with bounds (n)
b_s : rank-2 array('F') with bounds (n,nrhs) and b storage
x : rank-2 array('F') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('f') with bounds (nrhs)
berr : rank-1 array('f') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
overwrite_a : input int, optional
Default: 0
af : input rank-2 array('F') with bounds (n,n)
equed : input string(len=1), optional
Default: 'Y'
s : input rank-1 array('f') with bounds (n)
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zposvx`**(*a*, *b*[, *fact*, *af*, *equed*, *s*, *lower*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for `zposvx`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **a_s** : rank-2 array('D') with bounds (n,n) and a storage
lu : rank-2 array('D') with bounds (n,n) and af storage
equed : string(len=1)
s : rank-1 array('d') with bounds (n)
b_s : rank-2 array('D') with bounds (n,nrhs) and b storage
x : rank-2 array('D') with bounds (n,nrhs)
rcond : float
ferr : rank-1 array('d') with bounds (nrhs)
berr : rank-1 array('d') with bounds (nrhs)
info : int

Other Parameters
fact : input string(len=1), optional
Default: 'E'
overwrite_a : input int, optional

Default: 0
af : input rank-2 array('D') with bounds (n,n)
equed : input string(len=1), optional
Default: 'Y'
s : input rank-1 array('d') with bounds (n)
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`spocon`**(*a*, *anorm*[, *uplo*]) **= <fortran object>**
Wrapper for `spocon`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
uplo : input string(len=1), optional
Default: 'U'

`scipy.linalg.lapack.`**`dpocon`**(*a*, *anorm*[, *uplo*]) **= <fortran object>**
Wrapper for `dpocon`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
uplo : input string(len=1), optional
Default: 'U'

`scipy.linalg.lapack.`**`cpocon`**(*a*, *anorm*[, *uplo*]) **= <fortran object>**
Wrapper for `cpocon`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
uplo : input string(len=1), optional
Default: 'U'

`scipy.linalg.lapack.`**`zpocon`**(*a*, *anorm*[, *uplo*]) **= <fortran object>**
Wrapper for `zpocon`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
anorm : input float
Returns **rcond** : float
info : int
Other Parameters
uplo : input string(len=1), optional
Default: 'U'

`scipy.linalg.lapack.`**`spotrf`**(*a*[, *lower*, *clean*, *overwrite_a*]) **= <fortran object>**
Wrapper for `spotrf`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)

Returns **c** : rank-2 array('f') with bounds (n,n) and a storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lower : input int, optional
Default: 0
clean : input int, optional
Default: 1

scipy.linalg.lapack.**dpotrf**(*a*[, *lower*, *clean*, *overwrite_a*]) **= <fortran object>**
Wrapper for `dpotrf`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
Returns **c** : rank-2 array('d') with bounds (n,n) and a storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lower : input int, optional
Default: 0
clean : input int, optional
Default: 1

scipy.linalg.lapack.**cpotrf**(*a*[, *lower*, *clean*, *overwrite_a*]) **= <fortran object>**
Wrapper for `cpotrf`.

Parameters **a** : input rank-2 array('F') with bounds (n,n)
Returns **c** : rank-2 array('F') with bounds (n,n) and a storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lower : input int, optional
Default: 0
clean : input int, optional
Default: 1

scipy.linalg.lapack.**zpotrf**(*a*[, *lower*, *clean*, *overwrite_a*]) **= <fortran object>**
Wrapper for `zpotrf`.

Parameters **a** : input rank-2 array('D') with bounds (n,n)
Returns **c** : rank-2 array('D') with bounds (n,n) and a storage
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lower : input int, optional
Default: 0
clean : input int, optional
Default: 1

scipy.linalg.lapack.**spotri**(*c*[, *lower*, *overwrite_c*]) **= <fortran object>**
Wrapper for `spotri`.

Parameters **c** : input rank-2 array('f') with bounds (n,n)
Returns **inv_a** : rank-2 array('f') with bounds (n,n) and c storage
info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dpotri`**(*c*[, *lower*, *overwrite_c*]) **= <fortran object>**

Wrapper for `dpotri`.

Parameters **c** : input rank-2 array('d') with bounds (n,n)

Returns **inv_a** : rank-2 array('d') with bounds (n,n) and c storage
info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cpotri`**(*c*[, *lower*, *overwrite_c*]) **= <fortran object>**

Wrapper for `cpotri`.

Parameters **c** : input rank-2 array('F') with bounds (n,n)

Returns **inv_a** : rank-2 array('F') with bounds (n,n) and c storage
info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zpotri`**(*c*[, *lower*, *overwrite_c*]) **= <fortran object>**

Wrapper for `zpotri`.

Parameters **c** : input rank-2 array('D') with bounds (n,n)

Returns **inv_a** : rank-2 array('D') with bounds (n,n) and c storage
info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`spotrs`**(*c*, *b*[, *lower*, *overwrite_b*]) **= <fortran object>**

Wrapper for `spotrs`.

Parameters **c** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns **x** : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int

Other Parameters

overwrite_b : input int, optional
Default: 0

lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dpotrs`**(*c*, *b*[, *lower*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dpotrs`.

Parameters **c** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns **x** : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cpotrs`**(*c*, *b*[, *lower*, *overwrite_b*]) **= <fortran object>**
Wrapper for `cpotrs`.

Parameters **c** : input rank-2 array('F') with bounds (n,n)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns **x** : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zpotrs`**(*c*, *b*[, *lower*, *overwrite_b*]) **= <fortran object>**
Wrapper for `zpotrs`.

Parameters **c** : input rank-2 array('D') with bounds (n,n)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **x** : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_b : input int, optional
Default: 0
lower : input int, optional
Default: 0

`scipy.linalg.lapack.`**`crot`**(*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) **= <fortran object>**
Wrapper for `crot`.

Parameters **x** : input rank-1 array('F') with bounds (*)
y : input rank-1 array('F') with bounds (*)
c : input float
s : input complex

Returns **x** : rank-1 array('F') with bounds (*)
y : rank-1 array('F') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0

offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

`scipy.linalg.lapack.zrot`(*x*, *y*, *c*, *s*[, *n*, *offx*, *incx*, *offy*, *incy*, *overwrite_x*, *overwrite_y*]) **= <fortran object>**

Wrapper for `zrot`.

Parameters **x** : input rank-1 array('D') with bounds (*)
y : input rank-1 array('D') with bounds (*)
c : input float
s : input complex

Returns **x** : rank-1 array('D') with bounds (*)
y : rank-1 array('D') with bounds (*)

Other Parameters
n : input int, optional
Default: (len(x)-1-offx)/abs(incx)+1
overwrite_x : input int, optional
Default: 0
offx : input int, optional
Default: 0
incx : input int, optional
Default: 1
overwrite_y : input int, optional
Default: 0
offy : input int, optional
Default: 0
incy : input int, optional
Default: 1

`scipy.linalg.lapack.strsyl`(*a*, *b*, *c*[, *trana*, *tranb*, *isgn*, *overwrite_c*]) **= <fortran object>**

Wrapper for `strsyl`.

Parameters **a** : input rank-2 array('f') with bounds (m,m)
b : input rank-2 array('f') with bounds (n,n)
c : input rank-2 array('f') with bounds (m,n)

Returns **x** : rank-2 array('f') with bounds (m,n) and c storage
scale : float
info : int

Other Parameters
trana : input string(len=1), optional
Default: 'N'
tranb : input string(len=1), optional
Default: 'N'
isgn : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.dtrsyl`(*a*, *b*, *c*[, *trana*, *tranb*, *isgn*, *overwrite_c*]) **= <fortran object>**

Wrapper for `dtrsyl`.

Parameters **a** : input rank-2 array('d') with bounds (m,m)
b : input rank-2 array('d') with bounds (n,n)
c : input rank-2 array('d') with bounds (m,n)

Returns **x** : rank-2 array('d') with bounds (m,n) and c storage
scale : float
info : int

Other Parameters
trana : input string(len=1), optional
Default: 'N'
tranb : input string(len=1), optional
Default: 'N'
isgn : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

scipy.linalg.lapack.**ctrsyl**(*a*, *b*, *c*[, *trana*, *tranb*, *isgn*, *overwrite_c*]) **= <fortran object>**
Wrapper for `ctrsyl`.

Parameters **a** : input rank-2 array('F') with bounds (m,m)
b : input rank-2 array('F') with bounds (n,n)
c : input rank-2 array('F') with bounds (m,n)

Returns **x** : rank-2 array('F') with bounds (m,n) and c storage
scale : float
info : int

Other Parameters
trana : input string(len=1), optional
Default: 'N'
tranb : input string(len=1), optional
Default: 'N'
isgn : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

scipy.linalg.lapack.**ztrsyl**(*a*, *b*, *c*[, *trana*, *tranb*, *isgn*, *overwrite_c*]) **= <fortran object>**
Wrapper for `ztrsyl`.

Parameters **a** : input rank-2 array('D') with bounds (m,m)
b : input rank-2 array('D') with bounds (n,n)
c : input rank-2 array('D') with bounds (m,n)

Returns **x** : rank-2 array('D') with bounds (m,n) and c storage
scale : float
info : int

Other Parameters
trana : input string(len=1), optional
Default: 'N'
tranb : input string(len=1), optional
Default: 'N'
isgn : input int, optional
Default: 1
overwrite_c : input int, optional
Default: 0

scipy.linalg.lapack.**strtri**(*c*[, *lower*, *unitdiag*, *overwrite_c*]) **= <fortran object>**
Wrapper for `strtri`.

Parameters **c** : input rank-2 array('f') with bounds (n,n)

Returns **inv_c** : rank-2 array('f') with bounds (n,n) and c storage
info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

scipy.linalg.lapack.**dtrtri**(*c*[, *lower*, *unitdiag*, *overwrite_c*]) = <fortran object>

Wrapper for dtrtri.

Parameters **c** : input rank-2 array('d') with bounds (n,n)

Returns **inv_c** : rank-2 array('d') with bounds (n,n) and c storage

info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

scipy.linalg.lapack.**ctrtri**(*c*[, *lower*, *unitdiag*, *overwrite_c*]) = <fortran object>

Wrapper for ctrtri.

Parameters **c** : input rank-2 array('F') with bounds (n,n)

Returns **inv_c** : rank-2 array('F') with bounds (n,n) and c storage

info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

scipy.linalg.lapack.**ztrtri**(*c*[, *lower*, *unitdiag*, *overwrite_c*]) = <fortran object>

Wrapper for ztrtri.

Parameters **c** : input rank-2 array('D') with bounds (n,n)

Returns **inv_c** : rank-2 array('D') with bounds (n,n) and c storage

info : int

Other Parameters

overwrite_c : input int, optional
Default: 0

lower : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

scipy.linalg.lapack.**strtrs**(*a*, *b*[, *lower*, *trans*, *unitdiag*, *lda*, *overwrite_b*]) = <fortran object>

Wrapper for strtrs.

Parameters **a** : input rank-2 array('f') with bounds (lda,n)

b : input rank-2 array('f') with bounds (ldb,nrhs)

Returns **x** : rank-2 array('f') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

lower : input int, optional
Default: 0

trans : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

lda : input int, optional
Default: shape(a,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dtrtrs`**(*a*, *b*[, *lower*, *trans*, *unitdiag*, *lda*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dtrtrs`.

Parameters

a : input rank-2 array('d') with bounds (lda,n)

b : input rank-2 array('d') with bounds (ldb,nrhs)

Returns

x : rank-2 array('d') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

lower : input int, optional
Default: 0

trans : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

lda : input int, optional
Default: shape(a,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`ctrtrs`**(*a*, *b*[, *lower*, *trans*, *unitdiag*, *lda*, *overwrite_b*]) **= <fortran object>**

Wrapper for `ctrtrs`.

Parameters

a : input rank-2 array('F') with bounds (lda,n)

b : input rank-2 array('F') with bounds (ldb,nrhs)

Returns

x : rank-2 array('F') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

lower : input int, optional
Default: 0

trans : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

lda : input int, optional
Default: shape(a,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`ztrtrs`**(*a*, *b*[, *lower*, *trans*, *unitdiag*, *lda*, *overwrite_b*]) **= <fortran object>**

Wrapper for `ztrtrs`.

Parameters

a : input rank-2 array('D') with bounds (lda,n)

b : input rank-2 array('D') with bounds (ldb,nrhs)

Returns

x : rank-2 array('D') with bounds (ldb,nrhs) and b storage

info : int

Other Parameters

lower : input int, optional
Default: 0

trans : input int, optional
Default: 0

unitdiag : input int, optional
Default: 0

lda : input int, optional
Default: shape(a,0)

overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cunghr`**(*a*, *tau*[, *lo*, *hi*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cunghr`.

Parameters

a : input rank-2 array('F') with bounds (n,n)

tau : input rank-1 array('F') with bounds (n - 1)

Returns

ht : rank-2 array('F') with bounds (n,n) and a storage

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: hi-lo

`scipy.linalg.lapack.`**`zunghr`**(*a*, *tau*[, *lo*, *hi*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `zunghr`.

Parameters

a : input rank-2 array('D') with bounds (n,n)

tau : input rank-1 array('D') with bounds (n - 1)

Returns

ht : rank-2 array('D') with bounds (n,n) and a storage

info : int

Other Parameters

lo : input int, optional
Default: 0

hi : input int, optional
Default: n-1

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: hi-lo

`scipy.linalg.lapack.`**`cungqr`**(*a*, *tau*[, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `cungqr`.

Parameters

a : input rank-2 array('F') with bounds (m,n)

tau : input rank-1 array('F') with bounds (k)

Returns

q : rank-2 array('F') with bounds (m,n) and a storage

work : rank-1 array('F') with bounds (MAX(lwork,1))

info : int

Other Parameters

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 3*n

`scipy.linalg.lapack.`**`zungqr`**(*a*, *tau*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for `zungqr`.

Parameters **a** : input rank-2 array('D') with bounds (m,n)
tau : input rank-1 array('D') with bounds (k)

Returns **q** : rank-2 array('D') with bounds (m,n) and a storage
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

`scipy.linalg.lapack.`**`cungrq`**(*a*, *tau*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for `cungrq`.

Parameters **a** : input rank-2 array('F') with bounds (m,n)
tau : input rank-1 array('F') with bounds (k)

Returns **q** : rank-2 array('F') with bounds (m,n) and a storage
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

`scipy.linalg.lapack.`**`zungrq`**(*a*, *tau*[, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for `zungrq`.

Parameters **a** : input rank-2 array('D') with bounds (m,n)
tau : input rank-1 array('D') with bounds (k)

Returns **q** : rank-2 array('D') with bounds (m,n) and a storage
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

`scipy.linalg.lapack.`**`cunmqr`**(*side*, *trans*, *a*, *tau*, *c*, *lwork*[, *overwrite_c*]) = **<fortran object>**
Wrapper for `cunmqr`.

Parameters **side** : input string(len=1)
trans : input string(len=1)
a : input rank-2 array('F') with bounds (lda,k)
tau : input rank-1 array('F') with bounds (k)
c : input rank-2 array('F') with bounds (ldc,n)
lwork : input int

Returns **cq** : rank-2 array('F') with bounds (ldc,n) and c storage
work : rank-1 array('F') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zunmqr`**(*side*, *trans*, *a*, *tau*, *c*, *lwork*[, *overwrite_c*]) **= <fortran object>**

Wrapper for `zunmqr`.

Parameters
side : input string(len=1)
trans : input string(len=1)
a : input rank-2 array('D') with bounds (lda,k)
tau : input rank-1 array('D') with bounds (k)
c : input rank-2 array('D') with bounds (ldc,n)
lwork : input int

Returns
cq : rank-2 array('D') with bounds (ldc,n) and c storage
work : rank-1 array('D') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`sgtsv`**(*dl*, *d*, *du*, *b*[, *overwrite_dl*, *overwrite_d*, *overwrite_du*, *overwrite_b*]) **= <fortran object>**

Wrapper for `sgtsv`.

Parameters
dl : input rank-1 array('f') with bounds (n - 1)
d : input rank-1 array('f') with bounds (n)
du : input rank-1 array('f') with bounds (n - 1)
b : input rank-2 array('f') with bounds (n,nrhs)

Returns
du2 : rank-1 array('f') with bounds (n - 1) and dl storage
d : rank-1 array('f') with bounds (n)
du : rank-1 array('f') with bounds (n - 1)
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_dl : input int, optional
Default: 0
overwrite_d : input int, optional
Default: 0
overwrite_du : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dgtsv`**(*dl*, *d*, *du*, *b*[, *overwrite_dl*, *overwrite_d*, *overwrite_du*, *overwrite_b*]) **= <fortran object>**

Wrapper for `dgtsv`.

Parameters
dl : input rank-1 array('d') with bounds (n - 1)
d : input rank-1 array('d') with bounds (n)
du : input rank-1 array('d') with bounds (n - 1)
b : input rank-2 array('d') with bounds (n,nrhs)

Returns
du2 : rank-1 array('d') with bounds (n - 1) and dl storage
d : rank-1 array('d') with bounds (n)
du : rank-1 array('d') with bounds (n - 1)
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int

Other Parameters

overwrite_dl : input int, optional
Default: 0
overwrite_d : input int, optional
Default: 0
overwrite_du : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`cgtsv`**(*dl, d, du, b*[, *overwrite_dl, overwrite_d, overwrite_du, overwrite_b*]) = **<fortran object>**

Wrapper for `cgtsv`.

Parameters
dl : input rank-1 array('F') with bounds (n - 1)
d : input rank-1 array('F') with bounds (n)
du : input rank-1 array('F') with bounds (n - 1)
b : input rank-2 array('F') with bounds (n,nrhs)

Returns
du2 : rank-1 array('F') with bounds (n - 1) and dl storage
d : rank-1 array('F') with bounds (n)
du : rank-1 array('F') with bounds (n - 1)
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_dl : input int, optional
Default: 0
overwrite_d : input int, optional
Default: 0
overwrite_du : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.`**`zgtsv`**(*dl, d, du, b*[, *overwrite_dl, overwrite_d, overwrite_du, overwrite_b*]) = **<fortran object>**

Wrapper for `zgtsv`.

Parameters
dl : input rank-1 array('D') with bounds (n - 1)
d : input rank-1 array('D') with bounds (n)
du : input rank-1 array('D') with bounds (n - 1)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns
du2 : rank-1 array('D') with bounds (n - 1) and dl storage
d : rank-1 array('D') with bounds (n)
du : rank-1 array('D') with bounds (n - 1)
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_dl : input int, optional
Default: 0
overwrite_d : input int, optional
Default: 0
overwrite_du : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**sptsv**(*d*, *e*, *b*[, *overwrite_d*, *overwrite_e*, *overwrite_b*]) = **<fortran object>**
Wrapper for `sptsv`.

Parameters **d** : input rank-1 array('f') with bounds (n)
e : input rank-1 array('f') with bounds (n - 1)
b : input rank-2 array('f') with bounds (n,nrhs)
Returns **d** : rank-1 array('f') with bounds (n)
du : rank-1 array('f') with bounds (n - 1) and e storage
x : rank-2 array('f') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_d : input int, optional
Default: 0
overwrite_e : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**dptsv**(*d*, *e*, *b*[, *overwrite_d*, *overwrite_e*, *overwrite_b*]) = **<fortran object>**
Wrapper for `dptsv`.

Parameters **d** : input rank-1 array('d') with bounds (n)
e : input rank-1 array('d') with bounds (n - 1)
b : input rank-2 array('d') with bounds (n,nrhs)
Returns **d** : rank-1 array('d') with bounds (n)
du : rank-1 array('d') with bounds (n - 1) and e storage
x : rank-2 array('d') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_d : input int, optional
Default: 0
overwrite_e : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**cptsv**(*d*, *e*, *b*[, *overwrite_d*, *overwrite_e*, *overwrite_b*]) = **<fortran object>**
Wrapper for `cptsv`.

Parameters **d** : input rank-1 array('f') with bounds (n)
e : input rank-1 array('F') with bounds (n - 1)
b : input rank-2 array('F') with bounds (n,nrhs)
Returns **d** : rank-1 array('f') with bounds (n)
du : rank-1 array('F') with bounds (n - 1) and e storage
x : rank-2 array('F') with bounds (n,nrhs) and b storage
info : int
Other Parameters
overwrite_d : input int, optional
Default: 0
overwrite_e : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**zptsv**(*d*, *e*, *b*[, *overwrite_d*, *overwrite_e*, *overwrite_b*]) = **<fortran object>**
Wrapper for `zptsv`.

Parameters **d** : input rank-1 array('d') with bounds (n)
e : input rank-1 array('D') with bounds (n - 1)
b : input rank-2 array('D') with bounds (n,nrhs)

Returns **d** : rank-1 array('d') with bounds (n)
du : rank-1 array('D') with bounds (n - 1) and e storage
x : rank-2 array('D') with bounds (n,nrhs) and b storage
info : int

Other Parameters
overwrite_d : input int, optional
Default: 0
overwrite_e : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

`scipy.linalg.lapack.slamch`(*cmach*) = **<fortran slamch>**
Wrapper for `slamch`.

Parameters **cmach** : input string(len=1)

Returns **slamch** : float

`scipy.linalg.lapack.dlamch`(*cmach*) = **<fortran dlamch>**
Wrapper for `dlamch`.

Parameters **cmach** : input string(len=1)

Returns **dlamch** : float

`scipy.linalg.lapack.sorghr`(*a*, *tau*[, *lo*, *hi*, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for `sorghr`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
tau : input rank-1 array('f') with bounds (n - 1)

Returns **ht** : rank-2 array('f') with bounds (n,n) and a storage
info : int

Other Parameters
lo : input int, optional
Default: 0
hi : input int, optional
Default: n-1
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: hi-lo

`scipy.linalg.lapack.dorghr`(*a*, *tau*[, *lo*, *hi*, *lwork*, *overwrite_a*]) = **<fortran object>**
Wrapper for `dorghr`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
tau : input rank-1 array('d') with bounds (n - 1)

Returns **ht** : rank-2 array('d') with bounds (n,n) and a storage
info : int

Other Parameters
lo : input int, optional
Default: 0
hi : input int, optional
Default: n-1
overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: hi-lo

scipy.linalg.lapack.**sorgqr**(*a*, *tau*[, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for sorgqr.

Parameters **a** : input rank-2 array('f') with bounds (m,n)
tau : input rank-1 array('f') with bounds (k)
Returns **q** : rank-2 array('f') with bounds (m,n) and a storage
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

scipy.linalg.lapack.**dorgqr**(*a*, *tau*[, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for dorgqr.

Parameters **a** : input rank-2 array('d') with bounds (m,n)
tau : input rank-1 array('d') with bounds (k)
Returns **q** : rank-2 array('d') with bounds (m,n) and a storage
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*n

scipy.linalg.lapack.**sorgrq**(*a*, *tau*[, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for sorgrq.

Parameters **a** : input rank-2 array('f') with bounds (m,n)
tau : input rank-1 array('f') with bounds (k)
Returns **q** : rank-2 array('f') with bounds (m,n) and a storage
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

scipy.linalg.lapack.**dorgrq**(*a*, *tau*[, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for dorgrq.

Parameters **a** : input rank-2 array('d') with bounds (m,n)
tau : input rank-1 array('d') with bounds (k)
Returns **q** : rank-2 array('d') with bounds (m,n) and a storage
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int
Other Parameters
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: 3*m

`scipy.linalg.lapack.`**`sormqr`**(*side*, *trans*, *a*, *tau*, *c*, *lwork*[, *overwrite_c*]) **= <fortran object>**
Wrapper for `sormqr`.

Parameters **side** : input string(len=1)
trans : input string(len=1)
a : input rank-2 array('f') with bounds (lda,k)
tau : input rank-1 array('f') with bounds (k)
c : input rank-2 array('f') with bounds (ldc,n)
lwork : input int

Returns **cq** : rank-2 array('f') with bounds (ldc,n) and c storage
work : rank-1 array('f') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`dormqr`**(*side*, *trans*, *a*, *tau*, *c*, *lwork*[, *overwrite_c*]) **= <fortran object>**
Wrapper for `dormqr`.

Parameters **side** : input string(len=1)
trans : input string(len=1)
a : input rank-2 array('d') with bounds (lda,k)
tau : input rank-1 array('d') with bounds (k)
c : input rank-2 array('d') with bounds (ldc,n)
lwork : input int

Returns **cq** : rank-2 array('d') with bounds (ldc,n) and c storage
work : rank-1 array('d') with bounds (MAX(lwork,1))
info : int

Other Parameters
overwrite_c : input int, optional
Default: 0

`scipy.linalg.lapack.`**`ssbev`**(*ab*[, *compute_v*, *lower*, *ldab*, *overwrite_ab*]) **= <fortran object>**
Wrapper for `ssbev`.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

Returns **w** : rank-1 array('f') with bounds (n)
z : rank-2 array('f') with bounds (ldz,ldz)
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 1
compute_v : input int, optional
Default: 1
lower : input int, optional
Default: 0
ldab : input int, optional
Default: shape(ab,0)

`scipy.linalg.lapack.`**`dsbev`**(*ab*[, *compute_v*, *lower*, *ldab*, *overwrite_ab*]) **= <fortran object>**
Wrapper for `dsbev`.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)

Returns **w** : rank-1 array('d') with bounds (n)
z : rank-2 array('d') with bounds (ldz,ldz)
info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

scipy.linalg.lapack.**ssbevd**(*ab*[, *compute_v*, *lower*, *ldab*, *liwork*, *overwrite_ab*]) = **<fortran object>**

Wrapper for ssbevd.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

Returns **w** : rank-1 array('f') with bounds (n)

z : rank-2 array('f') with bounds (ldz,ldz)

info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

liwork : input int, optional
Default: (compute_v?3+5*n:1)

scipy.linalg.lapack.**dsbevd**(*ab*[, *compute_v*, *lower*, *ldab*, *liwork*, *overwrite_ab*]) = **<fortran object>**

Wrapper for dsbevd.

Parameters **ab** : input rank-2 array('d') with bounds (ldab,n)

Returns **w** : rank-1 array('d') with bounds (n)

z : rank-2 array('d') with bounds (ldz,ldz)

info : int

Other Parameters

overwrite_ab : input int, optional
Default: 1

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

ldab : input int, optional
Default: shape(ab,0)

liwork : input int, optional
Default: (compute_v?3+5*n:1)

scipy.linalg.lapack.**ssbevx**(*ab*, *vl*, *vu*, *il*, *iu*[, *ldab*, *compute_v*, *range*, *lower*, *abstol*, *mmax*, *overwrite_ab*]) = **<fortran object>**

Wrapper for ssbevx.

Parameters **ab** : input rank-2 array('f') with bounds (ldab,n)

vl : input float

vu : input float

il : input int
iu : input int

Returns
w : rank-1 array('f') with bounds (n)
z : rank-2 array('f') with bounds (ldz,mmax)
m : int
ifail : rank-1 array('i') with bounds ((compute_v?n:1))
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 1
ldab : input int, optional
Default: shape(ab,0)
compute_v : input int, optional
Default: 1
range : input int, optional
Default: 0
lower : input int, optional
Default: 0
abstol : input float, optional
Default: 0.0
mmax : input int, optional
Default: (compute_v?(range==2?(iu-il+1):n):1)

`scipy.linalg.lapack.`**`dsbevx`**(*ab*, *vl*, *vu*, *il*, *iu*[, *ldab*, *compute_v*, *range*, *lower*, *abstol*, *mmax*, *overwrite_ab*]) **= <fortran object>**

Wrapper for `dsbevx`.

Parameters
ab : input rank-2 array('d') with bounds (ldab,n)
vl : input float
vu : input float
il : input int
iu : input int

Returns
w : rank-1 array('d') with bounds (n)
z : rank-2 array('d') with bounds (ldz,mmax)
m : int
ifail : rank-1 array('i') with bounds ((compute_v?n:1))
info : int

Other Parameters
overwrite_ab : input int, optional
Default: 1
ldab : input int, optional
Default: shape(ab,0)
compute_v : input int, optional
Default: 1
range : input int, optional
Default: 0
lower : input int, optional
Default: 0
abstol : input float, optional
Default: 0.0
mmax : input int, optional
Default: (compute_v?(range==2?(iu-il+1):n):1)

`scipy.linalg.lapack.`**`ssyev`**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) **= <fortran object>**

Wrapper for `ssyev`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)

Returns **w** : rank-1 array('f') with bounds (n)

v : rank-2 array('f') with bounds (n,n) and a storage

info : int

Other Parameters

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 3*n-1

scipy.linalg.lapack.**dsyev**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for dsyev.

Parameters **a** : input rank-2 array('d') with bounds (n,n)

Returns **w** : rank-1 array('d') with bounds (n)

v : rank-2 array('d') with bounds (n,n) and a storage

info : int

Other Parameters

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: 3*n-1

scipy.linalg.lapack.**ssyevd**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for ssyevd.

Parameters **a** : input rank-2 array('f') with bounds (n,n)

Returns **w** : rank-1 array('f') with bounds (n)

v : rank-2 array('f') with bounds (n,n) and a storage

info : int

Other Parameters

compute_v : input int, optional
Default: 1

lower : input int, optional
Default: 0

overwrite_a : input int, optional
Default: 0

lwork : input int, optional
Default: (compute_v?1+6*n+2*n*n:2*n+1)

scipy.linalg.lapack.**dsyevd**(*a*[, *compute_v*, *lower*, *lwork*, *overwrite_a*]) = <fortran object>

Wrapper for dsyevd.

Parameters **a** : input rank-2 array('d') with bounds (n,n)

Returns **w** : rank-1 array('d') with bounds (n)

v : rank-2 array('d') with bounds (n,n) and a storage

info : int

Other Parameters

compute_v : input int, optional

Default: 1
lower : input int, optional
Default: 0
overwrite_a : input int, optional
Default: 0
lwork : input int, optional
Default: (compute_v?1+6*n+2*n*n:2*n+1)

scipy.linalg.lapack.**ssyevr**(*a*[, *jobz*, *range*, *uplo*, *il*, *iu*, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for `ssyevr`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
Returns **w** : rank-1 array('f') with bounds (n)
z : rank-2 array('f') with bounds (n,m)
info : int
Other Parameters
jobz : input string(len=1), optional
Default: 'V'
range : input string(len=1), optional
Default: 'A'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
il : input int, optional
Default: 1
iu : input int, optional
Default: n
lwork : input int, optional
Default: 26*n

scipy.linalg.lapack.**dsyevr**(*a*[, *jobz*, *range*, *uplo*, *il*, *iu*, *lwork*, *overwrite_a*]) = <fortran object>
Wrapper for `dsyevr`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
Returns **w** : rank-1 array('d') with bounds (n)
z : rank-2 array('d') with bounds (n,m)
info : int
Other Parameters
jobz : input string(len=1), optional
Default: 'V'
range : input string(len=1), optional
Default: 'A'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
il : input int, optional
Default: 1
iu : input int, optional
Default: n
lwork : input int, optional
Default: 26*n

scipy.linalg.lapack.**ssygv**(*a*, *b*[, *itype*, *jobz*, *uplo*, *overwrite_a*, *overwrite_b*]) = <fortran object>
Wrapper for `ssygv`.

Parameters
a : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,n)

Returns
a : rank-2 array('f') with bounds (n,n)
w : rank-1 array('f') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**dsygv**(*a*, *b*[, *itype*, *jobz*, *uplo*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for dsygv.

Parameters
a : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,n)

Returns
a : rank-2 array('d') with bounds (n,n)
w : rank-1 array('d') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0

scipy.linalg.lapack.**ssygvd**(*a*, *b*[, *itype*, *jobz*, *uplo*, *lwork*, *overwrite_a*, *overwrite_b*]) **= <fortran object>**

Wrapper for ssygvd.

Parameters
a : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,n)

Returns
a : rank-2 array('f') with bounds (n,n)
w : rank-1 array('f') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional

Default: 0
lwork : input int, optional
Default: 1+6*n+2*n*n

`scipy.linalg.lapack.`**`dsygvd`**(*a*, *b*[, *itype*, *jobz*, *uplo*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `dsygvd`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,n)

Returns **a** : rank-2 array('d') with bounds (n,n)
w : rank-1 array('d') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
lwork : input int, optional
Default: 1+6*n+2*n*n

`scipy.linalg.lapack.`**`ssygvx`**(*a*, *b*, *iu*[, *itype*, *jobz*, *uplo*, *il*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `ssygvx`.

Parameters **a** : input rank-2 array('f') with bounds (n,n)
b : input rank-2 array('f') with bounds (n,n)
iu : input int

Returns **w** : rank-1 array('f') with bounds (n)
z : rank-2 array('f') with bounds (n,m)
ifail : rank-1 array('i') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
il : input int, optional
Default: 1
lwork : input int, optional
Default: 8*n

`scipy.linalg.lapack.`**`dsygvx`**(*a*, *b*, *iu*[, *itype*, *jobz*, *uplo*, *il*, *lwork*, *overwrite_a*, *overwrite_b*]) = **<fortran object>**

Wrapper for `dsygvx`.

Parameters **a** : input rank-2 array('d') with bounds (n,n)
b : input rank-2 array('d') with bounds (n,n)
iu : input int

Returns **w** : rank-1 array('d') with bounds (n)
z : rank-2 array('d') with bounds (n,m)
ifail : rank-1 array('i') with bounds (n)
info : int

Other Parameters
itype : input int, optional
Default: 1
jobz : input string(len=1), optional
Default: 'V'
uplo : input string(len=1), optional
Default: 'L'
overwrite_a : input int, optional
Default: 0
overwrite_b : input int, optional
Default: 0
il : input int, optional
Default: 1
lwork : input int, optional
Default: 8*n

`scipy.linalg.lapack.`**`slange`**(*norm*, *a*) **= <fortran slange>**
Wrapper for `slange`.

Parameters **norm** : input string(len=1)
a : input rank-2 array('f') with bounds (m,n)

Returns **n2** : float

`scipy.linalg.lapack.`**`dlange`**(*norm*, *a*) **= <fortran dlange>**
Wrapper for `dlange`.

Parameters **norm** : input string(len=1)
a : input rank-2 array('d') with bounds (m,n)

Returns **n2** : float

`scipy.linalg.lapack.`**`clange`**(*norm*, *a*) **= <fortran clange>**
Wrapper for `clange`.

Parameters **norm** : input string(len=1)
a : input rank-2 array('F') with bounds (m,n)

Returns **n2** : float

`scipy.linalg.lapack.`**`zlange`**(*norm*, *a*) **= <fortran zlange>**
Wrapper for `zlange`.

Parameters **norm** : input string(len=1)
a : input rank-2 array('D') with bounds (m,n)

Returns **n2** : float

`scipy.linalg.lapack.`**`ilaver`** **= <fortran object>**
Wrapper for `ilaver`.

Returns **major** : int
minor : int
patch : int

5.12 BLAS Functions for Cython

Usable from Cython via:

```
cimport scipy.linalg.cython_blas
```

These wrappers do not check for alignment of arrays. Alignment should be checked before these wrappers are used.

Raw function pointers (Fortran-style pointer arguments):

- caxpy
- ccopy
- cdotc
- cdotu
- cgbmv
- cgemm
- cgemv
- cgerc
- cgeru
- chbmv
- chemm
- chemv
- cher
- cher2
- cher2k
- cherk
- chpmv
- chpr
- chpr2
- crotg
- cscal
- csrot
- csscal
- cswap
- csymm
- csyr2k
- csyrk
- ctbmv
- ctbsv
- ctpmv

- ctpsv
- ctrmm
- ctrmv
- ctrsm
- ctrsv
- dasum
- daxpy
- dcabs1
- dcopy
- ddot
- dgbmv
- dgemm
- dgemv
- dger
- dnrm2
- drot
- drotg
- drotm
- drotmg
- dsbmv
- dscal
- dsdot
- dspmv
- dspr
- dspr2
- dswap
- dsymm
- dsymv
- dsyr
- dsyr2
- dsyr2k
- dsyrk
- dtbmv
- dtbsv
- dtpmv
- dtpsv

- dtrmm
- dtrmv
- dtrsm
- dtrsv
- dzasum
- dznrm2
- icamax
- idamax
- isamax
- izamax
- lsame
- sasum
- saxpy
- scasum
- scnrm2
- scopy
- sdot
- sdsdot
- sgbmv
- sgemm
- sgemv
- sger
- snrm2
- srot
- srotg
- srotm
- srotmg
- ssbmv
- sscal
- sspmv
- sspr
- sspr2
- sswap
- ssymm
- ssymv
- ssyr

- ssyr2
- ssyr2k
- ssyrk
- stbmv
- stbsv
- stpmv
- stpsv
- strmm
- strmv
- strsm
- strsv
- zaxpy
- zcopy
- zdotc
- zdotu
- zdrot
- zdscal
- zgbmv
- zgemm
- zgemv
- zgerc
- zgeru
- zhbmv
- zhemm
- zhemv
- zher
- zher2
- zher2k
- zherk
- zhpmv
- zhpr
- zhpr2
- zrotg
- zscal
- zswap
- zsymm

- zsyr2k
- zsyrk
- ztbmv
- ztbsv
- ztpmv
- ztpsv
- ztrmm
- ztrmv
- ztrsm
- ztrsv

5.13 LAPACK functions for Cython

Usable from Cython via:

```
cimport scipy.linalg.cython_lapack
```

This module provides Cython-level wrappers for all primary routines included in LAPACK 3.1.0 except for `zcgesv` since its interface is not consistent from LAPACK 3.1.0 to 3.6.0. It also provides some of the fixed-api auxiliary routines.

These wrappers do not check for alignment of arrays. Alignment should be checked before these wrappers are used.

Raw function pointers (Fortran-style pointer arguments):

- cbdsqr
- cgbbrd
- cgbcon
- cgbequ
- cgbrfs
- cgbsv
- cgbsvx
- cgbtf2
- cgbtrf
- cgbtrs
- cgebak
- cgebal
- cgebd2
- cgebrd
- cgecon
- cgeequ
- cgees

- cgeesx
- cgeev
- cgeevx
- cgehd2
- cgehrd
- cgelq2
- cgelqf
- cgels
- cgelsd
- cgelss
- cgelsy
- cgeql2
- cgeqlf
- cgeqp3
- cgeqr2
- cgeqrf
- cgerfs
- cgerq2
- cgerqf
- cgesc2
- cgesdd
- cgesv
- cgesvd
- cgesvx
- cgetc2
- cgetf2
- cgetrf
- cgetri
- cgetrs
- cggbak
- cggbal
- cgges
- cggesx
- cggev
- cggevx
- cggglm

- cgghrd
- cgglse
- cggqrf
- cggrqf
- cgtcon
- cgtrfs
- cgtsv
- cgtsvx
- cgttrf
- cgttrs
- cgtts2
- chbev
- chbevd
- chbevx
- chbgst
- chbgv
- chbgvd
- chbgvx
- chbtrd
- checon
- cheev
- cheevd
- cheevr
- cheevx
- chegs2
- chegst
- chegv
- chegvd
- chegvx
- cherfs
- chesv
- chesvx
- chetd2
- chetf2
- chetrd
- chetrf

- chetri
- chetrs
- chgeqz
- chpcon
- chpev
- chpevd
- chpevx
- chpgst
- chpgv
- chpgvd
- chpgvx
- chprfs
- chpsv
- chpsvx
- chptrd
- chptrf
- chptri
- chptrs
- chsein
- chseqr
- clabrd
- clacgv
- clacn2
- clacon
- clacp2
- clacpy
- clacrm
- clacrt
- cladiv
- claed0
- claed7
- claed8
- claein
- claesy
- claev2
- clag2z

- clags2
- clagtm
- clahef
- clahqr
- clahr2
- claic1
- clals0
- clalsa
- clalsd
- clangb
- clange
- clangt
- clanhb
- clanhe
- clanhp
- clanhs
- clanht
- clansb
- clansp
- clansy
- clantb
- clantp
- clantr
- clapll
- clapmt
- claqgb
- claqge
- claqhb
- claqhe
- claqhp
- claqp2
- claqps
- claqr0
- claqr1
- claqr2
- claqr3

- claqr4
- claqr5
- claqsb
- claqsp
- claqsy
- clar1v
- clar2v
- clarcm
- clarf
- clarfb
- clarfg
- clarft
- clarfx
- clargv
- clarnv
- clarrv
- clartg
- clartv
- clarz
- clarzb
- clarzt
- clascl
- claset
- clasr
- classq
- claswp
- clasyf
- clatbs
- clatdf
- clatps
- clatrd
- clatrs
- clatrz
- clauu2
- clauum
- cpbcon

- cpbequ
- cpbrfs
- cpbstf
- cpbsv
- cpbsvx
- cpbtf2
- cpbtrf
- cpbtrs
- cpocon
- cpoequ
- cporfs
- cposv
- cposvx
- cpotf2
- cpotrf
- cpotri
- cpotrs
- cppcon
- cppequ
- cpprfs
- cppsv
- cppsvx
- cpptrf
- cpptri
- cpptrs
- cptcon
- cpteqr
- cptrfs
- cptsv
- cptsvx
- cpttrf
- cpttrs
- cptts2
- crot
- cspcon
- cspmv

- cspr
- csprfs
- cspsv
- cspsvx
- csptrf
- csptri
- csptrs
- csrscl
- cstedc
- cstegr
- cstein
- cstemr
- csteqr
- csycon
- csymv
- csyr
- csyrfs
- csysv
- csysvx
- csytf2
- csytrf
- csytri
- csytrs
- ctbcon
- ctbrfs
- ctbtrs
- ctgevc
- ctgex2
- ctgexc
- ctgsen
- ctgsja
- ctgsna
- ctgsy2
- ctgsyl
- ctpcon
- ctprfs

- ctptri
- ctptrs
- ctrcon
- ctrevc
- ctrexc
- ctrrfs
- ctrsen
- ctrsna
- ctrsyl
- ctrti2
- ctrtri
- ctrtrs
- ctzrzf
- cung2l
- cung2r
- cungbr
- cunghr
- cungl2
- cunglq
- cungql
- cungqr
- cungr2
- cungrq
- cungtr
- cunm2l
- cunm2r
- cunmbr
- cunmhr
- cunml2
- cunmlq
- cunmql
- cunmqr
- cunmr2
- cunmr3
- cunmrq
- cunmrz

- cunmtr
- cupgtr
- cupmtr
- dbdsdc
- dbdsqr
- ddisna
- dgbbrd
- dgbcon
- dgbequ
- dgbrfs
- dgbsv
- dgbsvx
- dgbtf2
- dgbtrf
- dgbtrs
- dgebak
- dgebal
- dgebd2
- dgebrd
- dgecon
- dgeequ
- dgees
- dgeesx
- dgeev
- dgeevx
- dgehd2
- dgehrd
- dgelq2
- dgelqf
- dgels
- dgelsd
- dgelss
- dgelsy
- dgeql2
- dgeqlf
- dgeqp3

- dgeqr2
- dgeqrf
- dgerfs
- dgerq2
- dgerqf
- dgesc2
- dgesdd
- dgesv
- dgesvd
- dgesvx
- dgetc2
- dgetf2
- dgetrf
- dgetri
- dgetrs
- dggbak
- dggbal
- dgges
- dggesx
- dggev
- dggevx
- dggglm
- dgghrd
- dgglse
- dggqrf
- dggrqf
- dgtcon
- dgtrfs
- dgtsv
- dgtsvx
- dgttrf
- dgttrs
- dgtts2
- dhgeqz
- dhsein
- dhseqr

- disnan
- dlabad
- dlabrd
- dlacn2
- dlacon
- dlacpy
- dladiv
- dlae2
- dlaebz
- dlaed0
- dlaed1
- dlaed2
- dlaed3
- dlaed4
- dlaed5
- dlaed6
- dlaed7
- dlaed8
- dlaed9
- dlaeda
- dlaein
- dlaev2
- dlaexc
- dlag2
- dlag2s
- dlags2
- dlagtf
- dlagtm
- dlagts
- dlagv2
- dlahqr
- dlahr2
- dlaic1
- dlaln2
- dlals0
- dlalsa

- dlalsd
- dlamch
- dlamrg
- dlaneg
- dlangb
- dlange
- dlangt
- dlanhs
- dlansb
- dlansp
- dlanst
- dlansy
- dlantb
- dlantp
- dlantr
- dlanv2
- dlapll
- dlapmt
- dlapy2
- dlapy3
- dlaqgb
- dlaqge
- dlaqp2
- dlaqps
- dlaqr0
- dlaqr1
- dlaqr2
- dlaqr3
- dlaqr4
- dlaqr5
- dlaqsb
- dlaqsp
- dlaqsy
- dlaqtr
- dlar1v
- dlar2v

- dlarf
- dlarfb
- dlarfg
- dlarft
- dlarfx
- dlargv
- dlarnv
- dlarra
- dlarrb
- dlarrc
- dlarrd
- dlarre
- dlarrf
- dlarrj
- dlarrk
- dlarrr
- dlarrv
- dlartg
- dlartv
- dlaruv
- dlarz
- dlarzb
- dlarzt
- dlas2
- dlascl
- dlasd0
- dlasd1
- dlasd2
- dlasd3
- dlasd4
- dlasd5
- dlasd6
- dlasd7
- dlasd8
- dlasda
- dlasdq

- dlasdt
- dlaset
- dlasq1
- dlasq2
- dlasq6
- dlasr
- dlasrt
- dlassq
- dlasv2
- dlaswp
- dlasy2
- dlasyf
- dlatbs
- dlatdf
- dlatps
- dlatrd
- dlatrs
- dlatrz
- dlauu2
- dlauum
- dopgtr
- dopmtr
- dorg2l
- dorg2r
- dorgbr
- dorghr
- dorgl2
- dorglq
- dorgql
- dorgqr
- dorgr2
- dorgrq
- dorgtr
- dorm2l
- dorm2r
- dormbr

- dormhr
- dorml2
- dormlq
- dormql
- dormqr
- dormr2
- dormr3
- dormrq
- dormrz
- dormtr
- dpbcon
- dpbequ
- dpbrfs
- dpbstf
- dpbsv
- dpbsvx
- dpbtf2
- dpbtrf
- dpbtrs
- dpocon
- dpoequ
- dporfs
- dposv
- dposvx
- dpotf2
- dpotrf
- dpotri
- dpotrs
- dppcon
- dppequ
- dpprfs
- dppsv
- dppsvx
- dpptrf
- dpptri
- dpptrs

- dptcon
- dpteqr
- dptrfs
- dptsv
- dptsvx
- dpttrf
- dpttrs
- dptts2
- drscl
- dsbev
- dsbevd
- dsbevx
- dsbgst
- dsbgv
- dsbgvd
- dsbgvx
- dsbtrd
- dsgesv
- dspcon
- dspev
- dspevd
- dspevx
- dspgst
- dspgv
- dspgvd
- dspgvx
- dsprfs
- dspsv
- dspsvx
- dsptrd
- dsptrf
- dsptri
- dsptrs
- dstebz
- dstedc
- dstegr

- dstein
- dstemr
- dsteqr
- dsterf
- dstev
- dstevd
- dstevr
- dstevx
- dsycon
- dsyev
- dsyevd
- dsyevr
- dsyevx
- dsygs2
- dsygst
- dsygv
- dsygvd
- dsygvx
- dsyrfs
- dsysv
- dsysvx
- dsytd2
- dsytf2
- dsytrd
- dsytrf
- dsytri
- dsytrs
- dtbcon
- dtbrfs
- dtbtrs
- dtgevc
- dtgex2
- dtgexc
- dtgsen
- dtgsja
- dtgsna

- dtgsy2
- dtgsyl
- dtpcon
- dtprfs
- dtptri
- dtptrs
- dtrcon
- dtrevc
- dtrexc
- dtrrfs
- dtrsen
- dtrsna
- dtrsyl
- dtrti2
- dtrtri
- dtrtrs
- dtzrzf
- dzsum1
- icmax1
- ieeeck
- ilaver
- izmax1
- sbdsdc
- sbdsqr
- scsum1
- sdisna
- sgbbrd
- sgbcon
- sgbequ
- sgbrfs
- sgbsv
- sgbsvx
- sgbtf2
- sgbtrf
- sgbtrs
- sgebak

- sgebal
- sgebd2
- sgebrd
- sgecon
- sgeequ
- sgees
- sgeesx
- sgeev
- sgeevx
- sgehd2
- sgehrd
- sgelq2
- sgelqf
- sgels
- sgelsd
- sgelss
- sgelsy
- sgeql2
- sgeqlf
- sgeqp3
- sgeqr2
- sgeqrf
- sgerfs
- sgerq2
- sgerqf
- sgesc2
- sgesdd
- sgesv
- sgesvd
- sgesvx
- sgetc2
- sgetf2
- sgetrf
- sgetri
- sgetrs
- sggbak

- sggbal
- sgges
- sggesx
- sggev
- sggevx
- sggglm
- sgghrd
- sgglse
- sggqrf
- sggrqf
- sgtcon
- sgtrfs
- sgtsv
- sgtsvx
- sgttrf
- sgttrs
- sgtts2
- shgeqz
- shsein
- shseqr
- slabad
- slabrd
- slacn2
- slacon
- slacpy
- sladiv
- slae2
- slaebz
- slaed0
- slaed1
- slaed2
- slaed3
- slaed4
- slaed5
- slaed6
- slaed7

- slaed8
- slaed9
- slaeda
- slaein
- slaev2
- slaexc
- slag2
- slag2d
- slags2
- slagtf
- slagtm
- slagts
- slagv2
- slahqr
- slahr2
- slaic1
- slaln2
- slals0
- slalsa
- slalsd
- slamch
- slamrg
- slangb
- slange
- slangt
- slanhs
- slansb
- slansp
- slanst
- slansy
- slantb
- slantp
- slantr
- slanv2
- slapll
- slapmt

- slapy2
- slapy3
- slaqgb
- slaqge
- slaqp2
- slaqps
- slaqr0
- slaqr1
- slaqr2
- slaqr3
- slaqr4
- slaqr5
- slaqsb
- slaqsp
- slaqsy
- slaqtr
- slar1v
- slar2v
- slarf
- slarfb
- slarfg
- slarft
- slarfx
- slargv
- slarnv
- slarra
- slarrb
- slarrc
- slarrd
- slarre
- slarrf
- slarrj
- slarrk
- slarrr
- slarrv
- slartg

- slartv
- slaruv
- slarz
- slarzb
- slarzt
- slas2
- slascl
- slasd0
- slasd1
- slasd2
- slasd3
- slasd4
- slasd5
- slasd6
- slasd7
- slasd8
- slasda
- slasdq
- slasdt
- slaset
- slasq1
- slasq2
- slasq6
- slasr
- slasrt
- slassq
- slasv2
- slaswp
- slasy2
- slasyf
- slatbs
- slatdf
- slatps
- slatrd
- slatrs
- slatrz

- slauu2
- slauum
- sopgtr
- sopmtr
- sorg2l
- sorg2r
- sorgbr
- sorghr
- sorgl2
- sorglq
- sorgql
- sorgqr
- sorgr2
- sorgrq
- sorgtr
- sorm2l
- sorm2r
- sormbr
- sormhr
- sorml2
- sormlq
- sormql
- sormqr
- sormr2
- sormr3
- sormrq
- sormrz
- sormtr
- spbcon
- spbequ
- spbrfs
- spbstf
- spbsv
- spbsvx
- spbtf2
- spbtrf

- spbtrs
- spocon
- spoequ
- sporfs
- sposv
- sposvx
- spotf2
- spotrf
- spotri
- spotrs
- sppcon
- sppequ
- spprfs
- sppsv
- sppsvx
- spptrf
- spptri
- spptrs
- sptcon
- spteqr
- sptrfs
- sptsv
- sptsvx
- spttrf
- spttrs
- sptts2
- srscl
- ssbev
- ssbevd
- ssbevx
- ssbgst
- ssbgv
- ssbgvd
- ssbgvx
- ssbtrd
- sspcon

- sspev
- sspevd
- sspevx
- sspgst
- sspgv
- sspgvd
- sspgvx
- ssprfs
- sspsv
- sspsvx
- ssptrd
- ssptrf
- ssptri
- ssptrs
- sstebz
- sstedc
- sstegr
- sstein
- sstemr
- ssteqr
- ssterf
- sstev
- sstevd
- sstevr
- sstevx
- ssycon
- ssyev
- ssyevd
- ssyevr
- ssyevx
- ssygs2
- ssygst
- ssygv
- ssygvd
- ssygvx
- ssyrfs

- ssysv
- ssysvx
- ssytd2
- ssytf2
- ssytrd
- ssytrf
- ssytri
- ssytrs
- stbcon
- stbrfs
- stbtrs
- stgevc
- stgex2
- stgexc
- stgsen
- stgsja
- stgsna
- stgsy2
- stgsyl
- stpcon
- stprfs
- stptri
- stptrs
- strcon
- strevc
- strexc
- strrfs
- strsen
- strsna
- strsyl
- strti2
- strtri
- strtrs
- stzrzf
- zbdsqr
- zdrscl

- zgbbrd
- zgbcon
- zgbequ
- zgbrfs
- zgbsv
- zgbsvx
- zgbtf2
- zgbtrf
- zgbtrs
- zgebak
- zgebal
- zgebd2
- zgebrd
- zgecon
- zgeequ
- zgees
- zgeesx
- zgeev
- zgeevx
- zgehd2
- zgehrd
- zgelq2
- zgelqf
- zgels
- zgelsd
- zgelss
- zgelsy
- zgeql2
- zgeqlf
- zgeqp3
- zgeqr2
- zgeqrf
- zgerfs
- zgerq2
- zgerqf
- zgesc2

- zgesdd
- zgesv
- zgesvd
- zgesvx
- zgetc2
- zgetf2
- zgetrf
- zgetri
- zgetrs
- zggbak
- zggbal
- zgges
- zggesx
- zggev
- zggevx
- zggglm
- zgghrd
- zgglse
- zggqrf
- zggrqf
- zgtcon
- zgtrfs
- zgtsv
- zgtsvx
- zgttrf
- zgttrs
- zgtts2
- zhbev
- zhbevd
- zhbevx
- zhbgst
- zhbgv
- zhbgvd
- zhbgvx
- zhbtrd
- zhecon

- zheev
- zheevd
- zheevr
- zheevx
- zhegs2
- zhegst
- zhegv
- zhegvd
- zhegvx
- zherfs
- zhesv
- zhesvx
- zhetd2
- zhetf2
- zhetrd
- zhetrf
- zhetri
- zhetrs
- zhgeqz
- zhpcon
- zhpev
- zhpevd
- zhpevx
- zhpgst
- zhpgv
- zhpgvd
- zhpgvx
- zhprfs
- zhpsv
- zhpsvx
- zhptrd
- zhptrf
- zhptri
- zhptrs
- zhsein
- zhseqr

- zlabrd
- zlacgv
- zlacn2
- zlacon
- zlacp2
- zlacpy
- zlacrm
- zlacrt
- zladiv
- zlaed0
- zlaed7
- zlaed8
- zlaein
- zlaesy
- zlaev2
- zlag2c
- zlags2
- zlagtm
- zlahef
- zlahqr
- zlahr2
- zlaic1
- zlals0
- zlalsa
- zlalsd
- zlangb
- zlange
- zlangt
- zlanhb
- zlanhe
- zlanhp
- zlanhs
- zlanht
- zlansb
- zlansp
- zlansy

- zlantb
- zlantp
- zlantr
- zlapll
- zlapmt
- zlaqgb
- zlaqge
- zlaqhb
- zlaqhe
- zlaqhp
- zlaqp2
- zlaqps
- zlaqr0
- zlaqr1
- zlaqr2
- zlaqr3
- zlaqr4
- zlaqr5
- zlaqsb
- zlaqsp
- zlaqsy
- zlar1v
- zlar2v
- zlarcm
- zlarf
- zlarfb
- zlarfg
- zlarft
- zlarfx
- zlargv
- zlarnv
- zlarrv
- zlartg
- zlartv
- zlarz
- zlarzb

- zlarzt
- zlascl
- zlaset
- zlasr
- zlassq
- zlaswp
- zlasyf
- zlatbs
- zlatdf
- zlatps
- zlatrd
- zlatrs
- zlatrz
- zlauu2
- zlauum
- zpbcon
- zpbequ
- zpbrfs
- zpbstf
- zpbsv
- zpbsvx
- zpbtf2
- zpbtrf
- zpbtrs
- zpocon
- zpoequ
- zporfs
- zposv
- zposvx
- zpotf2
- zpotrf
- zpotri
- zpotrs
- zppcon
- zppequ
- zpprfs

- zppsv
- zppsvx
- zpptrf
- zpptri
- zpptrs
- zptcon
- zpteqr
- zptrfs
- zptsv
- zptsvx
- zpttrf
- zpttrs
- zptts2
- zrot
- zspcon
- zspmv
- zspr
- zsprfs
- zspsv
- zspsvx
- zsptrf
- zsptri
- zsptrs
- zstedc
- zstegr
- zstein
- zstemr
- zsteqr
- zsycon
- zsymv
- zsyr
- zsyrfs
- zsysv
- zsysvx
- zsytf2
- zsytrf

- zsytri
- zsytrs
- ztbcon
- ztbrfs
- ztbtrs
- ztgevc
- ztgex2
- ztgexc
- ztgsen
- ztgsja
- ztgsna
- ztgsy2
- ztgsyl
- ztpcon
- ztprfs
- ztptri
- ztptrs
- ztrcon
- ztrevc
- ztrexc
- ztrrfs
- ztrsen
- ztrsna
- ztrsyl
- ztrti2
- ztrtri
- ztrtrs
- ztzrzf
- zung2l
- zung2r
- zungbr
- zunghr
- zungl2
- zunglq
- zungql
- zungqr

- zungr2
- zungrq
- zungtr
- zunm2l
- zunm2r
- zunmbr
- zunmhr
- zunml2
- zunmlq
- zunmql
- zunmqr
- zunmr2
- zunmr3
- zunmrq
- zunmrz
- zunmtr
- zupgtr
- zupmtr

5.14 Interpolative matrix decomposition (`scipy.linalg.interpolative`)

New in version 0.13.

An interpolative decomposition (ID) of a matrix $A \in \mathbb{C}^{m \times n}$ of rank $k \leq \min\{m, n\}$ is a factorization

$$A\Pi = \begin{bmatrix} A\Pi_1 & A\Pi_2 \end{bmatrix} = A\Pi_1 \begin{bmatrix} I & T \end{bmatrix},$$

where $\Pi = [\Pi_1, \Pi_2]$ is a permutation matrix with $\Pi_1 \in \{0, 1\}^{n \times k}$, i.e., $A\Pi_2 = A\Pi_1 T$. This can equivalently be written as $A = BP$, where $B = A\Pi_1$ and $P = [I, T]\Pi^{\mathsf{T}}$ are the *skeleton* and *interpolation matrices*, respectively.

If A does not have exact rank k, then there exists an approximation in the form of an ID such that $A = BP + E$, where $\|E\| \sim \sigma_{k+1}$ is on the order of the $(k + 1)$-th largest singular value of A. Note that σ_{k+1} is the best possible error for a rank-k approximation and, in fact, is achieved by the singular value decomposition (SVD) $A \approx USV^*$, where $U \in \mathbb{C}^{m \times k}$ and $V \in \mathbb{C}^{n \times k}$ have orthonormal columns and $S = \mathrm{diag}(\sigma_i) \in \mathbb{C}^{k \times k}$ is diagonal with nonnegative entries. The principal advantages of using an ID over an SVD are that:

- it is cheaper to construct;
- it preserves the structure of A; and
- it is more efficient to compute with in light of the identity submatrix of P.

5.14.1 Routines

Main functionality:

interp_decomp(A, eps_or_k[, rand])	Compute ID of a matrix.
reconstruct_matrix_from_id(B, idx, proj)	Reconstruct matrix from its ID.
reconstruct_interp_matrix(idx, proj)	Reconstruct interpolation matrix from ID.
reconstruct_skel_matrix(A, k, idx)	Reconstruct skeleton matrix from ID.
id_to_svd(B, idx, proj)	Convert ID to SVD.
svd(A, eps_or_k[, rand])	Compute SVD of a matrix via an ID.
estimate_spectral_norm(A[, its])	Estimate spectral norm of a matrix by the randomized power method.
estimate_spectral_norm_diff(A, B[, its])	Estimate spectral norm of the difference of two matrices by the randomized power method.
estimate_rank(A, eps)	Estimate matrix rank to a specified relative precision using randomized methods.

scipy.linalg.interpolative.**interp_decomp**(*A*, *eps_or_k*, *rand=True*)

Compute ID of a matrix.

An ID of a matrix *A* is a factorization defined by a rank *k*, a column index array *idx*, and interpolation coefficients *proj* such that:

```
numpy.dot(A[:,idx[:k]], proj) = A[:,idx[k:]]
```

The original matrix can then be reconstructed as:

```
numpy.hstack([A[:,idx[:k]],
                              numpy.dot(A[:,idx[:k]], proj)]
                         )[:,numpy.argsort(idx)]
```

or via the routine *reconstruct_matrix_from_id*. This can equivalently be written as:

```
numpy.dot(A[:,idx[:k]],
                    numpy.hstack([numpy.eye(k), proj])
                  )[:,np.argsort(idx)]
```

in terms of the skeleton and interpolation matrices:

```
B = A[:,idx[:k]]
```

and:

```
P = numpy.hstack([numpy.eye(k), proj])[:,np.argsort(idx)]
```

respectively. See also *reconstruct_interp_matrix* and *reconstruct_skel_matrix*.

The ID can be computed to any relative precision or rank (depending on the value of *eps_or_k*). If a precision is specified (*eps_or_k < 1*), then this function has the output signature:

```
k, idx, proj = interp_decomp(A, eps_or_k)
```

Otherwise, if a rank is specified (*eps_or_k >= 1*), then the output signature is:

```
idx, proj = interp_decomp(A, eps_or_k)
```

Parameters

A : numpy.ndarray or *scipy.sparse.linalg.LinearOperator* with *rmatvec*
  Matrix to be factored

eps_or_k : float or int

Relative error (if *eps_or_k* < *1*) or rank (if *eps_or_k* >= *1*) of approximation.

rand : bool, optional

Whether to use random sampling if *A* is of type `numpy.ndarray` (randomized algorithms are always used if *A* is of type `scipy.sparse.linalg.LinearOperator`).

Returns

k : int

Rank required to achieve specified relative precision if *eps_or_k* < *1*.

idx : `numpy.ndarray`

Column index array.

proj : `numpy.ndarray`

Interpolation coefficients.

`scipy.linalg.interpolative.`**`reconstruct_matrix_from_id`**(*B*, *idx*, *proj*)

Reconstruct matrix from its ID.

A matrix *A* with skeleton matrix *B* and ID indices and coefficients *idx* and *proj*, respectively, can be reconstructed as:

```
numpy.hstack([B, numpy.dot(B, proj)])[:,numpy.argsort(idx)]
```

See also `reconstruct_interp_matrix` and `reconstruct_skel_matrix`.

Parameters

B : `numpy.ndarray`

Skeleton matrix.

idx : `numpy.ndarray`

Column index array.

proj : `numpy.ndarray`

Interpolation coefficients.

Returns

`numpy.ndarray`

Reconstructed matrix.

`scipy.linalg.interpolative.`**`reconstruct_interp_matrix`**(*idx*, *proj*)

Reconstruct interpolation matrix from ID.

The interpolation matrix can be reconstructed from the ID indices and coefficients *idx* and *proj*, respectively, as:

```
P = numpy.hstack([numpy.eye(proj.shape[0]), proj])[:,numpy.argsort(idx)]
```

The original matrix can then be reconstructed from its skeleton matrix *B* via:

```
numpy.dot(B, P)
```

See also `reconstruct_matrix_from_id` and `reconstruct_skel_matrix`.

Parameters

idx : `numpy.ndarray`

Column index array.

proj : `numpy.ndarray`

Interpolation coefficients.

Returns

`numpy.ndarray`

Interpolation matrix.

`scipy.linalg.interpolative.`**`reconstruct_skel_matrix`**(*A*, *k*, *idx*)

Reconstruct skeleton matrix from ID.

The skeleton matrix can be reconstructed from the original matrix *A* and its ID rank and indices *k* and *idx*, respectively, as:

```
B = A[:,idx[:k]]
```

The original matrix can then be reconstructed via:

```
numpy.hstack([B, numpy.dot(B, proj)])[:,numpy.argsort(idx)]
```

See also `reconstruct_matrix_from_id` and `reconstruct_interp_matrix`.

Parameters
- **A** : `numpy.ndarray`
  Original matrix.
- **k** : int
  Rank of ID.
- **idx** : `numpy.ndarray`
  Column index array.

Returns
- `numpy.ndarray`
  Skeleton matrix.

`scipy.linalg.interpolative.`**`id_to_svd`**(*B*, *idx*, *proj*)

Convert ID to SVD.

The SVD reconstruction of a matrix with skeleton matrix *B* and ID indices and coefficients *idx* and *proj*, respectively, is:

```
U, S, V = id_to_svd(B, idx, proj)
A = numpy.dot(U, numpy.dot(numpy.diag(S), V.conj().T))
```

See also `svd`.

Parameters
- **B** : `numpy.ndarray`
  Skeleton matrix.
- **idx** : `numpy.ndarray`
  Column index array.
- **proj** : `numpy.ndarray`
  Interpolation coefficients.

Returns
- **U** : `numpy.ndarray`
  Left singular vectors.
- **S** : `numpy.ndarray`
  Singular values.
- **V** : `numpy.ndarray`
  Right singular vectors.

`scipy.linalg.interpolative.`**`svd`**(*A*, *eps_or_k*, *rand=True*)

Compute SVD of a matrix via an ID.

An SVD of a matrix *A* is a factorization:

```
A = numpy.dot(U, numpy.dot(numpy.diag(S), V.conj().T))
```

where *U* and *V* have orthonormal columns and *S* is nonnegative.

The SVD can be computed to any relative precision or rank (depending on the value of *eps_or_k*).

See also `interp_decomp` and `id_to_svd`.

Parameters
- **A** : `numpy.ndarray` or `scipy.sparse.linalg.LinearOperator`
  Matrix to be factored, given as either a `numpy.ndarray` or a `scipy.sparse.linalg.LinearOperator` with the *matvec* and *rmatvec* methods (to apply the matrix and its adjoint).
- **eps_or_k** : float or int

Relative error (if *eps_or_k < 1*) or rank (if *eps_or_k >= 1*) of approximation.

rand : bool, optional
Whether to use random sampling if *A* is of type numpy.ndarray (randomized algorithms are always used if *A* is of type *scipy.sparse.linalg.LinearOperator*).

Returns **U** : numpy.ndarray
Left singular vectors.

S : numpy.ndarray
Singular values.

V : numpy.ndarray
Right singular vectors.

scipy.linalg.interpolative.**estimate_spectral_norm**(*A*, *its=20*)

Estimate spectral norm of a matrix by the randomized power method.

Parameters **A** : *scipy.sparse.linalg.LinearOperator*
Matrix given as a *scipy.sparse.linalg.LinearOperator* with the *matvec* and *rmatvec* methods (to apply the matrix and its adjoint).

its : int, optional
Number of power method iterations.

Returns float
Spectral norm estimate.

scipy.linalg.interpolative.**estimate_spectral_norm_diff**(*A*, *B*, *its=20*)

Estimate spectral norm of the difference of two matrices by the randomized power method.

Parameters **A** : *scipy.sparse.linalg.LinearOperator*
First matrix given as a *scipy.sparse.linalg.LinearOperator* with the *matvec* and *rmatvec* methods (to apply the matrix and its adjoint).

B : *scipy.sparse.linalg.LinearOperator*
Second matrix given as a *scipy.sparse.linalg.LinearOperator* with the *matvec* and *rmatvec* methods (to apply the matrix and its adjoint).

its : int, optional
Number of power method iterations.

Returns float
Spectral norm estimate of matrix difference.

scipy.linalg.interpolative.**estimate_rank**(*A*, *eps*)

Estimate matrix rank to a specified relative precision using randomized methods.

The matrix *A* can be given as either a numpy.ndarray or a *scipy.sparse.linalg.LinearOperator*, with different algorithms used for each case. If *A* is of type numpy.ndarray, then the output rank is typically about 8 higher than the actual numerical rank.

Parameters **A** : numpy.ndarray or *scipy.sparse.linalg.LinearOperator*
Matrix whose rank is to be estimated, given as either a numpy.ndarray or a *scipy.sparse.linalg.LinearOperator* with the *rmatvec* method (to apply the matrix adjoint).

eps : float
Relative error for numerical rank definition.

Returns int
Estimated matrix rank.

Support functions:

seed([seed])	Seed the internal random number generator used in this ID package.

Continued on next page

Table 5.87 – continued from previous page

rand(*shape)	Generate standard uniform pseudorandom numbers via a very efficient lagged Fibonacci method.

`scipy.linalg.interpolative.`**`seed`**(*seed=None*)

Seed the internal random number generator used in this ID package.

The generator is a lagged Fibonacci method with 55-element internal state.

Parameters **seed** : int, sequence, 'default', optional

If 'default', the random seed is reset to a default value.

If *seed* is a sequence containing 55 floating-point numbers in range [0,1], these are used to set the internal state of the generator.

If the value is an integer, the internal state is obtained from `numpy.random.RandomState` (MT19937) with the integer used as the initial seed.

If *seed* is omitted (None), `numpy.random` is used to initialize the generator.

`scipy.linalg.interpolative.`**`rand`**(**shape*)

Generate standard uniform pseudorandom numbers via a very efficient lagged Fibonacci method.

This routine is used for all random number generation in this package and can affect ID and SVD results.

Parameters **shape**

Shape of output array

5.14.2 References

This module uses the ID software package *[R742]* by Martinsson, Rokhlin, Shkolnisky, and Tygert, which is a Fortran library for computing IDs using various algorithms, including the rank-revealing QR approach of *[R743]* and the more recent randomized methods described in *[R744]*, *[R745]*, and *[R746]*. This module exposes its functionality in a way convenient for Python users. Note that this module does not add any functionality beyond that of organizing a simpler and more consistent interface.

We advise the user to consult also the documentation for the ID package.

5.14.3 Tutorial

Initializing

The first step is to import *scipy.linalg.interpolative* by issuing the command:

```
>>> import scipy.linalg.interpolative as sli
```

Now let's build a matrix. For this, we consider a Hilbert matrix, which is well know to have low rank:

```
>>> from scipy.linalg import hilbert
>>> n = 1000
>>> A = hilbert(n)
```

We can also do this explicitly via:

```
>>> import numpy as np
>>> n = 1000
>>> A = np.empty((n, n), order='F')
>>> for j in range(n):
```

```
>>>     for i in range(m):
>>>         A[i,j] = 1. / (i + j + 1)
```

Note the use of the flag `order='F'` in `numpy.empty`. This instantiates the matrix in Fortran-contiguous order and is important for avoiding data copying when passing to the backend.

We then define multiplication routines for the matrix by regarding it as a *scipy.sparse.linalg.LinearOperator*:

```
>>> from scipy.sparse.linalg import aslinearoperator
>>> L = aslinearoperator(A)
```

This automatically sets up methods describing the action of the matrix and its adjoint on a vector.

Computing an ID

We have several choices of algorithm to compute an ID. These fall largely according to two dichotomies:

1. how the matrix is represented, i.e., via its entries or via its action on a vector; and
2. whether to approximate it to a fixed relative precision or to a fixed rank.

We step through each choice in turn below.

In all cases, the ID is represented by three parameters:

1. a rank `k`;
2. an index array `idx`; and
3. interpolation coefficients `proj`.

The ID is specified by the relation `np.dot(A[:,idx[:k]], proj) == A[:,idx[k:]]`.

From matrix entries

We first consider a matrix given in terms of its entries.

To compute an ID to a fixed precision, type:

```
>>> k, idx, proj = sli.interp_decomp(A, eps)
```

where `eps < 1` is the desired precision.

To compute an ID to a fixed rank, use:

```
>>> idx, proj = sli.interp_decomp(A, k)
```

where `k >= 1` is the desired rank.

Both algorithms use random sampling and are usually faster than the corresponding older, deterministic algorithms, which can be accessed via the commands:

```
>>> k, idx, proj = sli.interp_decomp(A, eps, rand=False)
```

and:

```
>>> idx, proj = sli.interp_decomp(A, k, rand=False)
```

respectively.

From matrix action

Now consider a matrix given in terms of its action on a vector as a *scipy.sparse.linalg.LinearOperator*.

To compute an ID to a fixed precision, type:

```
>>> k, idx, proj = sli.interp_decomp(L, eps)
```

To compute an ID to a fixed rank, use:

```
>>> idx, proj = sli.interp_decomp(L, k)
```

These algorithms are randomized.

Reconstructing an ID

The ID routines above do not output the skeleton and interpolation matrices explicitly but instead return the relevant information in a more compact (and sometimes more useful) form. To build these matrices, write:

```
>>> B = sli.reconstruct_skel_matrix(A, k, idx)
```

for the skeleton matrix and:

```
>>> P = sli.reconstruct_interp_matrix(idx, proj)
```

for the interpolation matrix. The ID approximation can then be computed as:

```
>>> C = np.dot(B, P)
```

This can also be constructed directly using:

```
>>> C = sli.reconstruct_matrix_from_id(B, idx, proj)
```

without having to first compute `P`.

Alternatively, this can be done explicitly as well using:

```
>>> B = A[:,idx[:k]]
>>> P = np.hstack([np.eye(k), proj])[:,np.argsort(idx)]
>>> C = np.dot(B, P)
```

Computing an SVD

An ID can be converted to an SVD via the command:

```
>>> U, S, V = sli.id_to_svd(B, idx, proj)
```

The SVD approximation is then:

```
>>> C = np.dot(U, np.dot(np.diag(S), np.dot(V.conj().T)))
```

The SVD can also be computed "fresh" by combining both the ID and conversion steps into one command. Following the various ID algorithms above, there are correspondingly various SVD algorithms that one can employ.

From matrix entries

We consider first SVD algorithms for a matrix given in terms of its entries.

To compute an SVD to a fixed precision, type:

```
>>> U, S, V = sli.svd(A, eps)
```

To compute an SVD to a fixed rank, use:

```
>>> U, S, V = sli.svd(A, k)
```

Both algorithms use random sampling; for the determinstic versions, issue the keyword `rand=False` as above.

From matrix action

Now consider a matrix given in terms of its action on a vector.

To compute an SVD to a fixed precision, type:

```
>>> U, S, V = sli.svd(L, eps)
```

To compute an SVD to a fixed rank, use:

```
>>> U, S, V = sli.svd(L, k)
```

Utility routines

Several utility routines are also available.

To estimate the spectral norm of a matrix, use:

```
>>> snorm = sli.estimate_spectral_norm(A)
```

This algorithm is based on the randomized power method and thus requires only matrix-vector products. The number of iterations to take can be set using the keyword `its` (default: `its=20`). The matrix is interpreted as a *scipy.sparse.linalg.LinearOperator*, but it is also valid to supply it as a `numpy.ndarray`, in which case it is trivially converted using *scipy.sparse.linalg.aslinearoperator*.

The same algorithm can also estimate the spectral norm of the difference of two matrices `A1` and `A2` as follows:

```
>>> diff = sli.estimate_spectral_norm_diff(A1, A2)
```

This is often useful for checking the accuracy of a matrix approximation.

Some routines in *scipy.linalg.interpolative* require estimating the rank of a matrix as well. This can be done with either:

```
>>> k = sli.estimate_rank(A, eps)
```

or:

```
>>> k = sli.estimate_rank(L, eps)
```

depending on the representation. The parameter `eps` controls the definition of the numerical rank.

Finally, the random number generation required for all randomized routines can be controlled via *scipy.linalg.interpolative.seed*. To reset the seed values to their original values, use:

```
>>> sli.seed('default')
```

To specify the seed values, use:

```
>>> sli.seed(s)
```

where `s` must be an integer or array of 55 floats. If an integer, the array of floats is obtained by using `np.random.rand` with the given integer seed.

To simply generate some random numbers, type:

```
>>> sli.rand(n)
```

where `n` is the number of random numbers to generate.

Remarks

The above functions all automatically detect the appropriate interface and work with both real and complex data types, passing input arguments to the proper backend routine.

5.15 Miscellaneous routines (scipy.misc)

Various utilities that don't have another home.

Note that Pillow (https://python-pillow.org/) is not a dependency of SciPy, but the image manipulation functions indicated in the list below are not available without it.

ascent()	Get an 8-bit grayscale bit-depth, 512 x 512 derived image for easy use in demos
bytescale(data[, cmin, cmax, high, low])	Byte scales an array (image).
central_diff_weights(Np[, ndiv])	Return weights for an Np-point central derivative.
comb(N, k[, exact, repetition])	The number of combinations of N things taken k at a time.
derivative(func, x0[, dx, n, args, order])	Find the n-th derivative of a function at a point.
face([gray])	Get a 1024 x 768, color image of a raccoon face.
factorial(n[, exact])	The factorial of a number or array of numbers.
factorial2(n[, exact])	Double factorial.
factorialk(n, k[, exact])	Multifactorial of n of order k, n(!!...!).
fromimage(im[, flatten, mode])	Return a copy of a PIL image as a numpy array.
imfilter(arr, ftype)	Simple filtering of an image.
imread(name[, flatten, mode])	Read an image from a file as an array.
imresize(arr, size[, interp, mode])	Resize an image.
imrotate(arr, angle[, interp])	Rotate an image counter-clockwise by angle degrees.
imsave(name, arr[, format])	Save an array as an image.
imshow(arr)	Simple showing of an image through an external viewer.
info([object, maxwidth, output, toplevel])	Get help information for a function, class, or module.
lena()	Function that previously returned an example image
logsumexp(a[, axis, b, keepdims, return_sign])	Compute the log of the sum of exponentials of input elements.
pade(an, m)	Return Pade approximation to a polynomial as the ratio of two polynomials.
toimage(arr[, high, low, cmin, cmax, pal, ...])	Takes a numpy array and returns a PIL image.

Continued on next page

Table 5.88 – continued from previous page

source(object[, output])	Print or write to a file the source code for a NumPy object.
who([vardict])	Print the NumPy arrays in the given dictionary.

scipy.misc.**ascent**()

Get an 8-bit grayscale bit-depth, 512 x 512 derived image for easy use in demos

The image is derived from accent-to-the-top.jpg at http://www.public-domain-image.com/people-public-domain-images-pictures/

Parameters **None**

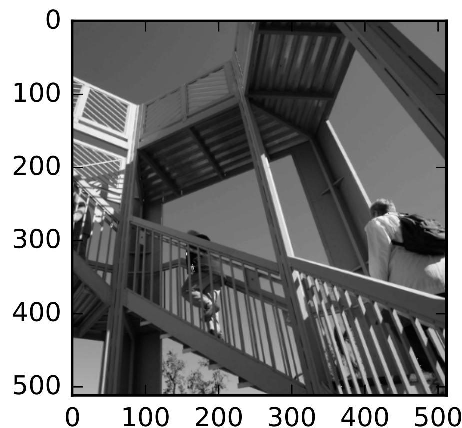
Returns **ascent** : ndarray

convenient image to use for testing and demonstration

Examples

```
>>> import scipy.misc
>>> ascent = scipy.misc.ascent()
>>> ascent.shape
(512, 512)
>>> ascent.max()
255
```

```
>>> import matplotlib.pyplot as plt
>>> plt.gray()
>>> plt.imshow(ascent)
>>> plt.show()
```



scipy.misc.**bytescale**(*data*, *cmin=None*, *cmax=None*, *high=255*, *low=0*)

Byte scales an array (image).

Byte scaling means converting the input image to uint8 dtype and scaling the range to `(low, high)` (default 0-255). If the input image already has dtype uint8, no scaling is done.

Parameters **data** : ndarray

PIL image data array.

cmin : scalar, optional

Bias scaling of small values. Default is `data.min()`.

cmax : scalar, optional
Bias scaling of large values. Default is `data.max()`.

high : scalar, optional
Scale max value to *high*. Default is 255.

low : scalar, optional
Scale min value to *low*. Default is 0.

Returns **img_array** : uint8 ndarray
The byte-scaled array.

Examples

```
>>> from scipy.misc import bytescale
>>> img = np.array([[ 91.06794177,   3.39058326,  84.4221549 ],
...                 [ 73.88003259,  80.91433048,   4.88878881],
...                 [ 51.53875334,  34.45808177,  27.5873488 ]])
>>> bytescale(img)
array([[255,   0, 236],
       [205, 225,   4],
       [140,  90,  70]], dtype=uint8)
>>> bytescale(img, high=200, low=100)
array([[200, 100, 192],
       [180, 188, 102],
       [155, 135, 128]], dtype=uint8)
>>> bytescale(img, cmin=0, cmax=255)
array([[91,  3, 84],
       [74, 81,  5],
       [52, 34, 28]], dtype=uint8)
```

`scipy.misc.`**`central_diff_weights`**(*Np*, *ndiv=1*)

Return weights for an Np-point central derivative.

Assumes equally-spaced function points.

If weights are in the vector w, then derivative is w[0] * f(x-ho*dx) + ... + w[-1] * f(x+h0*dx)

Parameters **Np** : int
Number of points for the central derivative.

ndiv : int, optional
Number of divisions. Default is 1.

Notes

Can be inaccurate for large number of points.

`scipy.misc.`**`comb`**(*N*, *k*, *exact=False*, *repetition=False*)

The number of combinations of N things taken k at a time.

This is often expressed as "N choose k".

Parameters **N** : int, ndarray
Number of things.

k : int, ndarray
Number of elements taken.

exact : bool, optional
If *exact* is False, then floating point precision is used, otherwise exact long integer is computed.

repetition : bool, optional
If *repetition* is True, then the number of combinations with repetition is computed.

Returns **val** : int, ndarray
The total number of combinations.

See also:

binom Binomial coefficient ufunc

Notes

- Array arguments accepted only for exact=False case.
- If k > N, N < 0, or k < 0, then a 0 is returned.

Examples

```
>>> from scipy.special import comb
>>> k = np.array([3, 4])
>>> n = np.array([10, 10])
>>> comb(n, k, exact=False)
array([ 120.,  210.])
>>> comb(10, 3, exact=True)
120L
>>> comb(10, 3, exact=True, repetition=True)
220L
```

scipy.misc.**derivative**(*func*, *x0*, *dx=1.0*, *n=1*, *args=()*, *order=3*)
Find the n-th derivative of a function at a point.

Given a function, use a central difference formula with spacing *dx* to compute the *n*-th derivative at *x0*.

Parameters **func** : function
Input function.
x0 : float
The point at which *n*-th derivative is found.
dx : float, optional
Spacing.
n : int, optional
Order of the derivative. Default is 1.
args : tuple, optional
Arguments
order : int, optional
Number of points to use, must be odd.

Notes

Decreasing the step size too small can result in round-off error.

Examples

```
>>> from scipy.misc import derivative
>>> def f(x):
...     return x**3 + x**2
>>> derivative(f, 1.0, dx=1e-6)
4.9999999999217337
```

scipy.misc.**face**(*gray=False*)
Get a 1024 x 768, color image of a raccoon face.

raccoon-procyon-lotor.jpg at http://www.public-domain-image.com

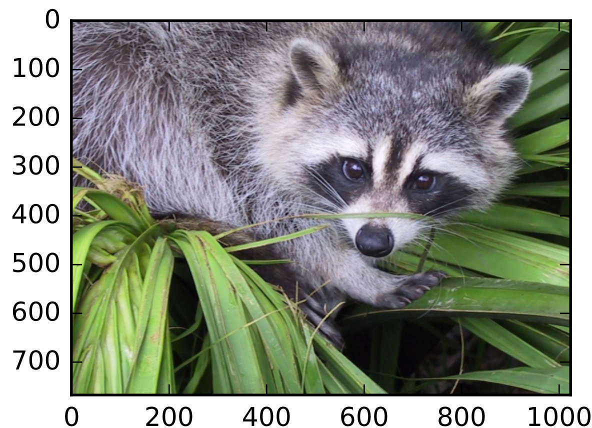
Parameters **gray** : bool, optional
If True return 8-bit grey-scale image, otherwise return a color image

Returns **face** : ndarray
image of a racoon face

Examples

```
>>> import scipy.misc
>>> face = scipy.misc.face()
>>> face.shape
(768, 1024, 3)
>>> face.max()
255
>>> face.dtype
dtype('uint8')
```

```
>>> import matplotlib.pyplot as plt
>>> plt.gray()
>>> plt.imshow(face)
>>> plt.show()
```



`scipy.misc.factorial(n, exact=False)`
The factorial of a number or array of numbers.

The factorial of non-negative integer *n* is the product of all positive integers less than or equal to *n*:

```
n! = n * (n - 1) * (n - 2) * ... * 1
```

Parameters **n** : int or array_like of ints
Input values. If `n < 0`, the return value is 0.

exact : bool, optional
If True, calculate the answer exactly using long integer arithmetic. If False, result is approximated in floating point rapidly using the *gamma* function. Default is False.

Returns **nf** : float or int or ndarray
Factorial of *n*, as integer or float depending on *exact*.

Notes

For arrays with `exact=True`, the factorial is computed only once, for the largest input, with each other result computed in the process. The output dtype is increased to `int64` or `object` if necessary.

With `exact=False` the factorial is approximated using the gamma function:

$$n! = \Gamma(n+1)$$

Examples

```
>>> from scipy.special import factorial
>>> arr = np.array([3, 4, 5])
>>> factorial(arr, exact=False)
array([   6.,   24.,  120.])
>>> factorial(arr, exact=True)
array([  6,  24, 120])
>>> factorial(5, exact=True)
120L
```

`scipy.misc.`**`factorial2`**(*n*, *exact=False*)

Double factorial.

This is the factorial with every second value skipped. E.g., `7!! = 7 * 5 * 3 * 1`. It can be approximated numerically as:

```
n!! = special.gamma(n/2+1)*2**((m+1)/2)/sqrt(pi)   n odd
    = 2**(n/2) * (n/2)!                            n even
```

Parameters

n : int or array_like

Calculate `n!!`. Arrays are only supported with *exact* set to False. If `n < 0`, the return value is 0.

exact : bool, optional

The result can be approximated rapidly using the gamma-formula above (default). If *exact* is set to True, calculate the answer exactly using integer arithmetic.

Returns

nff : float or int

Double factorial of *n*, as an int or a float depending on *exact*.

Examples

```
>>> from scipy.special import factorial2
>>> factorial2(7, exact=False)
array(105.00000000000001)
>>> factorial2(7, exact=True)
105L
```

`scipy.misc.`**`factorialk`**(*n*, *k*, *exact=True*)

Multifactorial of n of order k, n(!!...!).

This is the multifactorial of n skipping k values. For example,

factorialk(17, 4) = 17!!!! = 17 * 13 * 9 * 5 * 1

In particular, for any integer `n`, we have

factorialk(n, 1) = factorial(n)
factorialk(n, 2) = factorial2(n)

Parameters

n : int

Calculate multifactorial. If $n < 0$, the return value is 0.

k : int
Order of multifactorial.

exact : bool, optional
If exact is set to True, calculate the answer exactly using integer arithmetic.

Returns **val** : int
Multifactorial of *n*.

Raises **NotImplementedError**
Raises when exact is False

Examples

```
>>> from scipy.special import factorialk
>>> factorialk(5, 1, exact=True)
120L
>>> factorialk(5, 3, exact=True)
10L
```

`scipy.misc.`**`fromimage`**(*im*, *flatten=False*, *mode=None*)

Return a copy of a PIL image as a numpy array.

Parameters **im** : PIL image
Input image.

flatten : bool
If true, convert the output to grey-scale.

mode : str, optional
Mode to convert image to, e.g. `'RGB'`. See the Notes of the *`imread`* docstring for more details.

Returns **fromimage** : ndarray
The different colour bands/channels are stored in the third dimension, such that a grey-image is MxN, an RGB-image MxNx3 and an RGBA-image MxNx4.

`scipy.misc.`**`imfilter`**(*arr*, *ftype*)

Simple filtering of an image.

Parameters **arr** : ndarray
The array of Image in which the filter is to be applied.

ftype : str
The filter that has to be applied. Legal values are: 'blur', 'contour', 'detail', 'edge_enhance', 'edge_enhance_more', 'emboss', 'find_edges', 'smooth', 'smooth_more', 'sharpen'.

Returns **imfilter** : ndarray
The array with filter applied.

Raises **ValueError**
Unknown filter type. If the filter you are trying to apply is unsupported.

`scipy.misc.`**`imread`**(*name*, *flatten=False*, *mode=None*)

Read an image from a file as an array.

Parameters **name** : str or file object
The file name or file object to be read.

flatten : bool, optional
If True, flattens the color layers into a single gray-scale layer.

mode : str, optional
Mode to convert image to, e.g. `'RGB'`. See the Notes for more details.

Returns **imread** : ndarray
The array obtained by reading the image.

Notes

`imread` uses the Python Imaging Library (PIL) to read an image. The following notes are from the PIL documentation.

mode can be one of the following strings:

- 'L' (8-bit pixels, black and white)
- 'P' (8-bit pixels, mapped to any other mode using a color palette)
- 'RGB' (3x8-bit pixels, true color)
- 'RGBA' (4x8-bit pixels, true color with transparency mask)
- 'CMYK' (4x8-bit pixels, color separation)
- 'YCbCr' (3x8-bit pixels, color video format)
- 'I' (32-bit signed integer pixels)
- 'F' (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including 'LA' ('L' with alpha), 'RGBX' (true color with padding) and 'RGBa' (true color with premultiplied alpha).

When translating a color image to black and white (mode 'L', 'I' or 'F'), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

When *flatten* is True, the image is converted using mode 'F'. When *mode* is not None and *flatten* is True, the image is first converted according to *mode*, and the result is then flattened using mode 'F'.

`scipy.misc.`**`imresize`**(*arr*, *size*, *interp='bilinear'*, *mode=None*)

Resize an image.

Parameters

arr : ndarray
: The array of image to be resized.

size : int, float or tuple
- int - Percentage of current size.
- float - Fraction of current size.
- tuple - Size of the output image.

interp : str, optional
: Interpolation to use for re-sizing ('nearest', 'lanczos', 'bilinear', 'bicubic' or 'cubic').

mode : str, optional
: The PIL image mode ('P', 'L', etc.) to convert *arr* before resizing.

Returns

imresize : ndarray
: The resized array of image.

See also:

`toimage` Implicitly used to convert *arr* according to *mode*.

`scipy.ndimage.zoom`
More generic implementation that does not use PIL.

`scipy.misc.`**`imrotate`**(*arr*, *angle*, *interp='bilinear'*)

Rotate an image counter-clockwise by angle degrees.

Parameters

arr : ndarray
: Input array of image to be rotated.

angle : float

The angle of rotation.

interp : str, optional
    Interpolation
    - 'nearest' : for nearest neighbor
    - 'bilinear' : for bilinear
    - 'lanczos' : for lanczos
    - 'cubic' : for bicubic
    - 'bicubic' : for bicubic

Returns **imrotate** : ndarray
    The rotated array of image.

`scipy.misc.`**`imsave`**(*name*, *arr*, *format=None*)

Save an array as an image.

Parameters **name** : str or file object
    Output file name or file object.

arr : ndarray, MxN or MxNx3 or MxNx4
    Array containing image values. If the shape is `MxN`, the array represents a grey-level image. Shape `MxNx3` stores the red, green and blue bands along the last dimension. An alpha layer may be included, specified as the last colour band of an `MxNx4` array.

format : str
    Image format. If omitted, the format to use is determined from the file name extension. If a file object was used instead of a file name, this parameter should always be used.

Examples

Construct an array of gradient intensity values and save to file:

```
>>> from scipy.misc import imsave
>>> x = np.zeros((255, 255))
>>> x = np.zeros((255, 255), dtype=np.uint8)
>>> x[:] = np.arange(255)
>>> imsave('gradient.png', x)
```

Construct an array with three colour bands (R, G, B) and store to file:

```
>>> rgb = np.zeros((255, 255, 3), dtype=np.uint8)
>>> rgb[..., 0] = np.arange(255)
>>> rgb[..., 1] = 55
>>> rgb[..., 2] = 1 - np.arange(255)
>>> imsave('rgb_gradient.png', rgb)
```

`scipy.misc.`**`imshow`**(*arr*)

Simple showing of an image through an external viewer.

Uses the image viewer specified by the environment variable SCIPY_PIL_IMAGE_VIEWER, or if that is not defined then *see*, to view a temporary file generated from array data.

Parameters **arr** : ndarray
    Array of image data to show.

Returns None

Examples

```
>>> a = np.tile(np.arange(255), (255,1))
>>> from scipy import misc
>>> misc.imshow(a)
```

`scipy.misc.info`(*object=None*, *maxwidth=76*, *output=<open file '<stdout>', mode 'w'>*, *toplevel='scipy'*)

Get help information for a function, class, or module.

Parameters **object** : object or str, optional

Input object or name to get information about. If *object* is a numpy object, its docstring is given. If it is a string, available modules are searched for matching objects. If None, information about `info` itself is returned.

maxwidth : int, optional

Printing width.

output : file like object, optional

File like object that the output is written to, default is `stdout`. The object has to be opened in 'w' or 'a' mode.

toplevel : str, optional

Start search at this level.

See also:

source, `lookfor`

Notes

When used interactively with an object, `np.info(obj)` is equivalent to `help(obj)` on the Python prompt or `obj?` on the IPython prompt.

Examples

```
>>> np.info(np.polyval)
   polyval(p, x)
     Evaluate the polynomial p at x.
     ...
```

When using a string for *object* it is possible to get multiple results.

```
>>> np.info('fft')
     *** Found in numpy ***
Core FFT routines
...
     *** Found in numpy.fft ***
 fft(a, n=None, axis=-1)
...
     *** Repeat reference found in numpy.fft.fftpack ***
     *** Total of 3 references found. ***
```

`scipy.misc.lena`()

Function that previously returned an example image

Note: Removed in 0.17

Parameters **None**

Returns None

Raises **RuntimeError**

This functionality has been removed due to licensing reasons.

See also:

face, *ascent*

Notes

The image previously returned by this function has an incompatible license and has been removed from SciPy. Please use `face` or `ascent` instead.

`scipy.misc.logsumexp`(*a*, *axis=None*, *b=None*, *keepdims=False*, *return_sign=False*)
Compute the log of the sum of exponentials of input elements.

Parameters

a : array_like
Input array.

axis : None or int or tuple of ints, optional
Axis or axes over which the sum is taken. By default *axis* is None, and all elements are summed.
New in version 0.11.0.

keepdims : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array.
New in version 0.15.0.

b : array-like, optional
Scaling factor for exp(*a*) must be of the same shape as *a* or broadcastable to *a*. These values may be negative in order to implement subtraction.
New in version 0.12.0.

return_sign : bool, optional
If this is set to True, the result will be a pair containing sign information; if False, results that are negative will be returned as NaN. Default is False (no sign information).
New in version 0.16.0.

Returns

——-

res : ndarray
The result, `np.log(np.sum(np.exp(a)))` calculated in a numerically more stable way. If *b* is given then `np.log(np.sum(b*np.exp(a)))` is returned.

sgn : ndarray
If return_sign is True, this will be an array of floating-point numbers matching res and +1, 0, or -1 depending on the sign of the result. If False, only one result is returned.

See also:

`numpy.logaddexp`, `numpy.logaddexp2`

Notes

Numpy has a logaddexp function which is very similar to `logsumexp`, but only handles two arguments. *logaddexp.reduce* is similar to this function, but may be less stable.

Examples

```
>>> from scipy.special import logsumexp
>>> a = np.arange(10)
>>> np.log(np.sum(np.exp(a)))
9.4586297444267107
>>> logsumexp(a)
9.4586297444267107
```

With weights

```
>>> a = np.arange(10)
>>> b = np.arange(10, 0, -1)
>>> logsumexp(a, b=b)
```

```
9.9170178533034665
>>> np.log(np.sum(b*np.exp(a)))
9.9170178533034647
```

Returning a sign flag

```
>>> logsumexp([1,2],b=[1,-1],return_sign=True)
(1.5413248546129181, -1.0)
```

Notice that `logsumexp` does not directly support masked arrays. To use it on a masked array, convert the mask into zero weights:

```
>>> a = np.ma.array([np.log(2), 2, np.log(3)],
...                  mask=[False, True, False])
>>> b = (~a.mask).astype(int)
>>> logsumexp(a.data, b=b), np.log(5)
1.6094379124341005, 1.6094379124341005
```

`scipy.misc.`**`pade`**`(`*an*, *m*`)`

Return Pade approximation to a polynomial as the ratio of two polynomials.

Parameters **an** : (N,) array_like

Taylor series coefficients.

m : int

The order of the returned approximating polynomials.

Returns **p, q** : Polynomial class

The Pade approximation of the polynomial defined by *an* is `p(x)/q(x)`.

Examples

```
>>> from scipy.interpolate import pade
>>> e_exp = [1.0, 1.0, 1.0/2.0, 1.0/6.0, 1.0/24.0, 1.0/120.0]
>>> p, q = pade(e_exp, 2)
```

```
>>> e_exp.reverse()
>>> e_poly = np.poly1d(e_exp)
```

Compare `e_poly(x)` and the Pade approximation `p(x)/q(x)`

```
>>> e_poly(1)
2.7166666666666668
```

```
>>> p(1)/q(1)
2.7179487179487181
```

`scipy.misc.`**`toimage`**`(`*arr*, *high=255*, *low=0*, *cmin=None*, *cmax=None*, *pal=None*, *mode=None*, *channel_axis=None*`)`

Takes a numpy array and returns a PIL image.

The mode of the PIL image depends on the array shape and the *pal* and *mode* keywords.

For 2-D arrays, if *pal* is a valid (N,3) byte-array giving the RGB values (from 0 to 255) then `mode='P'`, otherwise `mode='L'`, unless mode is given as ‘F’ or ‘I’ in which case a float and/or integer array is made.

Notes

For 3-D arrays, the *channel_axis* argument tells which dimension of the array holds the channel data.

For 3-D arrays if one of the dimensions is 3, the mode is 'RGB' by default or 'YCbCr' if selected.

The numpy array must be either 2 dimensional or 3 dimensional.

`scipy.misc.source`(*object*, *output=<open file '<stdout>', mode 'w'>*)

Print or write to a file the source code for a NumPy object.

The source code is only returned for objects written in Python. Many functions and classes are defined in C and will therefore not return useful information.

Parameters **object** : numpy object

Input object. This can be any object (function, class, module, ...).

output : file object, optional

If *output* not supplied then source code is printed to screen (sys.stdout). File object must be created with either write 'w' or append 'a' modes.

See also:

`lookfor`, *info*

Examples

```
>>> np.source(np.interp)
In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
def interp(x, xp, fp, left=None, right=None):
    """.... (full docstring printed)"""
    if isinstance(x, (float, int, number)):
        return compiled_interp([x], xp, fp, left, right).item()
    else:
        return compiled_interp(x, xp, fp, left, right)
```

The source code is only returned for objects written in Python.

```
>>> np.source(np.array)
Not available for this object.
```

`scipy.misc.who`(*vardict=None*)

Print the NumPy arrays in the given dictionary.

If there is no dictionary passed in or *vardict* is None then returns NumPy arrays in the globals() dictionary (all NumPy arrays in the namespace).

Parameters **vardict** : dict, optional

A dictionary possibly containing ndarrays. Default is globals().

Returns **out** : None

Returns 'None'.

Notes

Prints out the name, shape, bytes and type of all of the ndarrays present in *vardict*.

Examples

```
>>> a = np.arange(10)
>>> b = np.ones(20)
>>> np.who()
Name            Shape            Bytes            Type
===========================================================
a               10               40               int32
b               20               160              float64
Upper bound on total bytes  =       200
```

```
>>> d = {'x': np.arange(2.0), 'y': np.arange(3.0), 'txt': 'Some str',
... 'idx':5}
>>> np.who(d)
Name            Shape            Bytes            Type
===========================================================
y               3                24               float64
x               2                16               float64
Upper bound on total bytes  =       40
```

5.16 Multi-dimensional image processing (`scipy.ndimage`)

This package contains various functions for multi-dimensional image processing.

5.16.1 Filters

convolve(input, weights[, output, mode, ...])	Multidimensional convolution.
convolve1d(input, weights[, axis, output, ...])	Calculate a one-dimensional convolution along the given axis.
correlate(input, weights[, output, mode, ...])	Multi-dimensional correlation.
correlate1d(input, weights[, axis, output, ...])	Calculate a one-dimensional correlation along the given axis.
gaussian_filter(input, sigma[, order, ...])	Multidimensional Gaussian filter.
gaussian_filter1d(input, sigma[, axis, ...])	One-dimensional Gaussian filter.
gaussian_gradient_magnitude(input, sigma[, ...])	Multidimensional gradient magnitude using Gaussian derivatives.
gaussian_laplace(input, sigma[, output, ...])	Multidimensional Laplace filter using gaussian second derivatives.
generic_filter(input, function[, size, ...])	Calculates a multi-dimensional filter using the given function.
generic_filter1d(input, function, filter_size)	Calculate a one-dimensional filter along the given axis.
generic_gradient_magnitude(input, derivative)	Gradient magnitude using a provided gradient function.
generic_laplace(input, derivative2[, ...])	N-dimensional Laplace filter using a provided second derivative function
laplace(input[, output, mode, cval])	N-dimensional Laplace filter based on approximate second derivatives.
maximum_filter(input[, size, footprint, ...])	Calculates a multi-dimensional maximum filter.
maximum_filter1d(input, size[, axis, ...])	Calculate a one-dimensional maximum filter along the given axis.
median_filter(input[, size, footprint, ...])	Calculates a multidimensional median filter.
minimum_filter(input[, size, footprint, ...])	Calculates a multi-dimensional minimum filter.
minimum_filter1d(input, size[, axis, ...])	Calculate a one-dimensional minimum filter along the given axis.
percentile_filter(input, percentile[, size, ...])	Calculates a multi-dimensional percentile filter.
prewitt(input[, axis, output, mode, cval])	Calculate a Prewitt filter.
rank_filter(input, rank[, size, footprint, ...])	Calculates a multi-dimensional rank filter.
sobel(input[, axis, output, mode, cval])	Calculate a Sobel filter.
uniform_filter(input[, size, output, mode, ...])	Multi-dimensional uniform filter.

Continued on next page

Table 5.89 – continued from previous page

uniform_filter1d(input, size[, axis, ...])	Calculate a one-dimensional uniform filter along the given axis.

`scipy.ndimage.convolve`(*input*, *weights*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)
Multidimensional convolution.

The array is convolved with the given kernel.

Parameters

input : array_like
Input array to filter.

weights : array_like
Array of weights, same number of dimensions as input

output : ndarray, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional
the *mode* parameter determines how the array borders are handled. For 'constant' mode, values beyond borders are set to be *cval*. Default is 'reflect'.

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : array_like, optional
The *origin* parameter controls the placement of the filter, relative to the centre of the current element of the input. Default of 0 is equivalent to `(0,)*input.ndim`.

Returns

result : ndarray
The result of convolution of *input* with *weights*.

See also:

`correlate` Correlate an image with a kernel.

Notes

Each value in result is $C_i = \sum_j I_{i+k-j} W_j$, where W is the *weights* kernel, j is the n-D spatial index over W, I is the *input* and k is the coordinate of the center of W, specified by *origin* in the input parameters.

Examples

Perhaps the simplest case to understand is `mode='constant', cval=0.0`, because in this case borders (i.e. where the *weights* kernel, centered on any one value, extends beyond an edge of *input*.

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> k = np.array([[1,1,1],[1,1,0],[1,0,0]])
>>> from scipy import ndimage
>>> ndimage.convolve(a, k, mode='constant', cval=0.0)
array([[11, 10,  7,  4],
       [10,  3, 11, 11],
       [15, 12, 14,  7],
       [12,  3,  7,  0]])
```

Setting `cval=1.0` is equivalent to padding the outer edge of *input* with 1.0's (and then extracting only the original region of the result).

```
>>> ndimage.convolve(a, k, mode='constant', cval=1.0)
array([[13, 11,  8,  7],
       [11,  3, 11, 14],
       [16, 12, 14, 10],
       [15,  6, 10,  5]])
```

With `mode='reflect'` (the default), outer values are reflected at the edge of *input* to fill in missing values.

```
>>> b = np.array([[2, 0, 0],
...               [1, 0, 0],
...               [0, 0, 0]])
>>> k = np.array([[0,1,0], [0,1,0], [0,1,0]])
>>> ndimage.convolve(b, k, mode='reflect')
array([[5, 0, 0],
       [3, 0, 0],
       [1, 0, 0]])
```

This includes diagonally at the corners.

```
>>> k = np.array([[1,0,0],[0,1,0],[0,0,1]])
>>> ndimage.convolve(b, k)
array([[4, 2, 0],
       [3, 2, 0],
       [1, 1, 0]])
```

With `mode='nearest'`, the single nearest value in to an edge in *input* is repeated as many times as needed to match the overlapping *weights*.

```
>>> c = np.array([[2, 0, 1],
...               [1, 0, 0],
...               [0, 0, 0]])
>>> k = np.array([[0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0],
...               [0, 1, 0]])
>>> ndimage.convolve(c, k, mode='nearest')
array([[7, 0, 3],
       [5, 0, 2],
       [3, 0, 1]])
```

`scipy.ndimage.`**`convolve1d`**(*input*, *weights*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculate a one-dimensional convolution along the given axis.

The lines of the array along the given axis are convolved with the given weights.

Parameters

input : array_like
  Input array to filter.

weights : ndarray
  One-dimensional sequence of numbers.

axis : int, optional
  The axis of *input* along which to calculate. Default is -1.

output : array, optional
  The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional

The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Returns **convolve1d** : ndarray
Convolved array with same shape as input

`scipy.ndimage.`**`correlate`**(*input*, *weights*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)
Multi-dimensional correlation.

The array is correlated with the given kernel.

Parameters **input** : array-like
input array to filter

weights : ndarray
array of weights, same number of dimensions as input

output : array, optional
The `output` parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional
The `mode` parameter determines how the array borders are handled, where `cval` is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if `mode` is 'constant'. Default is 0.0

origin : scalar, optional
The `origin` parameter controls the placement of the filter. Default 0

See also:

`convolve` Convolve an image with a kernel.

`scipy.ndimage.`**`correlate1d`**(*input*, *weights*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)
Calculate a one-dimensional correlation along the given axis.

The lines of the array along the given axis are correlated with the given weights.

Parameters **input** : array_like
Input array to filter.

weights : array
One-dimensional sequence of numbers.

axis : int, optional
The axis of *input* along which to calculate. Default is -1.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

`scipy.ndimage.gaussian_filter`(*input*, *sigma*, *order=0*, *output=None*, *mode='reflect'*, *cval=0.0*, *truncate=4.0*)

Multidimensional Gaussian filter.

Parameters

input : array_like
: Input array to filter.

sigma : scalar or sequence of scalars
: Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

order : {0, 1, 2, 3} or sequence from same set, optional
: The order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

output : array, optional
: The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
: The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
: Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

truncate : float
: Truncate the filter at this many standard deviations. Default is 4.0.

Returns

gaussian_filter : ndarray
: Returned array of same shape as *input*.

Notes

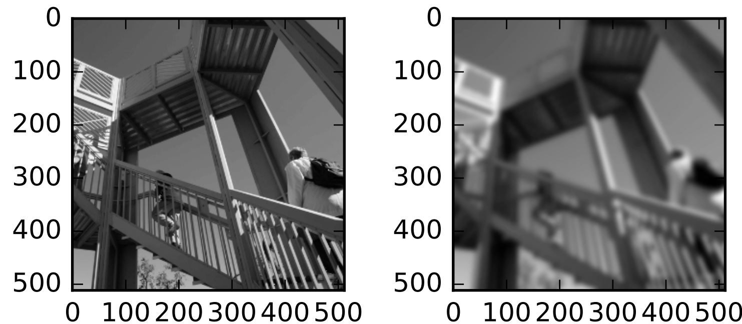
The multidimensional filter is implemented as a sequence of one-dimensional convolution filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Examples

```
>>> from scipy.ndimage import gaussian_filter
>>> a = np.arange(50, step=2).reshape((5,5))
>>> a
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38],
       [40, 42, 44, 46, 48]])
>>> gaussian_filter(a, sigma=1)
array([[ 4,  6,  8,  9, 11],
       [10, 12, 14, 15, 17],
       [20, 22, 24, 25, 27],
       [29, 31, 33, 34, 36],
       [35, 37, 39, 40, 42]])
```

```
>>> from scipy import misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
```

```
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = gaussian_filter(ascent, sigma=5)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



scipy.ndimage.**gaussian_filter1d**(*input*, *sigma*, *axis=-1*, *order=0*, *output=None*, *mode='reflect'*, *cval=0.0*, *truncate=4.0*)

One-dimensional Gaussian filter.

Parameters

input : array_like
Input array to filter.

sigma : scalar
standard deviation for Gaussian kernel

axis : int, optional
The axis of *input* along which to calculate. Default is -1.

order : {0, 1, 2, 3}, optional
An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

truncate : float, optional
Truncate the filter at this many standard deviations. Default is 4.0.

Returns

gaussian_filter1d : ndarray

scipy.ndimage.**gaussian_gradient_magnitude**(*input*, *sigma*, *output=None*, *mode='reflect'*, *cval=0.0*, ***kwargs*)

Multidimensional gradient magnitude using Gaussian derivatives.

Parameters

input : array_like
Input array to filter.

sigma : scalar or sequence of scalars
The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes..

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

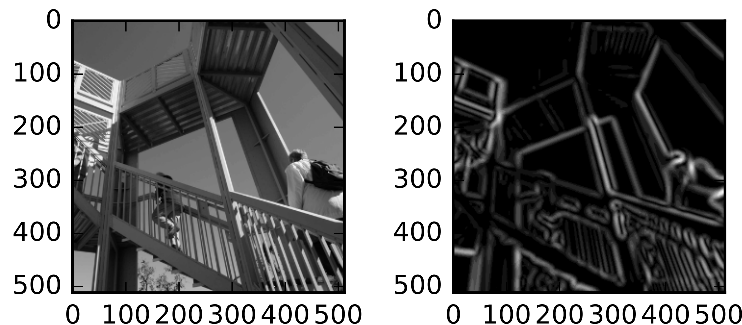
Extra keyword arguments will be passed to gaussian_filter().

Returns

gaussian_gradient_magnitude : ndarray
Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.gaussian_gradient_magnitude(ascent, sigma=5)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.gaussian_laplace`(*input*, *sigma*, *output=None*, *mode='reflect'*, *cval=0.0*, ***kwargs*)

Multidimensional Laplace filter using gaussian second derivatives.

Parameters

input : array_like
Input array to filter.

sigma : scalar or sequence of scalars
The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

Extra keyword arguments will be passed to gaussian_filter().

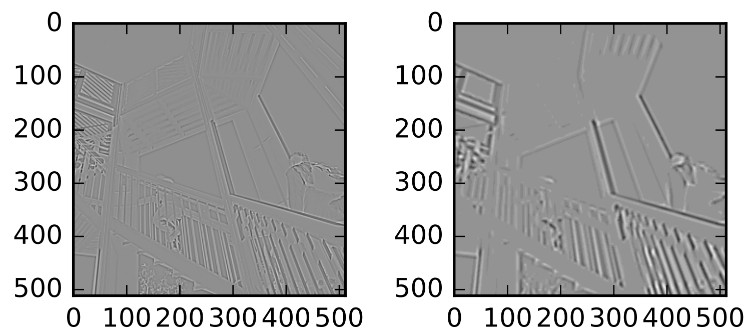
Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> ascent = misc.ascent()
```

```
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
```

```
>>> result = ndimage.gaussian_laplace(ascent, sigma=1)
>>> ax1.imshow(result)
```

```
>>> result = ndimage.gaussian_laplace(ascent, sigma=3)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.generic_filter(input, function, size=None, footprint=None, output=None, mode='reflect', cval=0.0, origin=0, extra_arguments=(), extra_keywords=None)`

Calculates a multi-dimensional filter using the given function.

At each element the provided function is called. The input values within the filter footprint at that element are passed to the function as a 1D array of double values.

Parameters

input : array_like
Input array to filter.

function : {callable, scipy.LowLevelCallable}
Function to apply at each element.

size : scalar or tuple, optional
See footprint, below

footprint : array, optional
Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

extra_arguments : sequence, optional
Sequence of extra positional arguments to pass to passed function

extra_keywords : dict, optional
dict of extra keyword arguments to pass to passed function

Notes

This function also accepts low-level callback functions with one of the following signatures and wrapped in *scipy.LowLevelCallable*:

```
int callback(double *buffer, npy_intp filter_size,
             double *return_value, void *user_data)
int callback(double *buffer, intptr_t filter_size,
             double *return_value, void *user_data)
```

The calling function iterates over the elements of the input and output arrays, calling the callback function at each element. The elements within the footprint of the filter at the current element are passed through the `buffer` parameter, and the number of elements within the footprint through `filter_size`. The calculated value is returned in `return_value`. `user_data` is the data pointer provided to *scipy.LowLevelCallable* as-is.

The callback function must return an integer error status that is zero if something went wrong and one otherwise. If an error occurs, you should normally set the python error status with an informative message before returning, otherwise a default error message is set by the calling function.

In addition, some other low-level function pointer specifications are accepted, but these are for backward compatibility only and should not be used in new code.

scipy.ndimage.**generic_filter1d**(*input*, *function*, *filter_size*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*, *extra_arguments=()*, *extra_keywords=None*)

Calculate a one-dimensional filter along the given axis.

`generic_filter1d` iterates over the lines of the array, calling the given function at each line. The arguments of the line are the input line, and the output line. The input and output lines are 1D double arrays. The input line is extended appropriately according to the filter size and origin. The output line must be modified in-place with the result.

Parameters

input : array_like
Input array to filter.

function : {callable, scipy.LowLevelCallable}
Function to apply along given axis.

filter_size : scalar
Length of the filter.

axis : int, optional
The axis of *input* along which to calculate. Default is -1.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

extra_arguments : sequence, optional
Sequence of extra positional arguments to pass to passed function

extra_keywords : dict, optional
dict of extra keyword arguments to pass to passed function

Notes

This function also accepts low-level callback functions with one of the following signatures and wrapped in `scipy.LowLevelCallable`:

```
int function(double *input_line, npy_intp input_length,
             double *output_line, npy_intp output_length,
             void *user_data)
int function(double *input_line, intptr_t input_length,
             double *output_line, intptr_t output_length,
             void *user_data)
```

The calling function iterates over the lines of the input and output arrays, calling the callback function at each line. The current line is extended according to the border conditions set by the calling function, and the result is copied into the array that is passed through `input_line`. The length of the input line (after extension) is passed through `input_length`. The callback function should apply the filter and store the result in the array passed through `output_line`. The length of the output line is passed through `output_length`. `user_data` is the data pointer provided to `scipy.LowLevelCallable` as-is.

The callback function must return an integer error status that is zero if something went wrong and one otherwise. If an error occurs, you should normally set the python error status with an informative message before returning,

otherwise a default error message is set by the calling function.

In addition, some other low-level function pointer specifications are accepted, but these are for backward compatibility only and should not be used in new code.

`scipy.ndimage.generic_gradient_magnitude`(*input*, *derivative*, *output=None*, *mode='reflect'*, *cval=0.0*, *extra_arguments=()*, *extra_keywords=None*)

Gradient magnitude using a provided gradient function.

Parameters **input** : array_like

Input array to filter.

derivative : callable

Callable with the following signature:

```
derivative(input, axis, output, mode, cval,
           *extra_arguments, **extra_keywords)
```

See *extra_arguments*, *extra_keywords* below. *derivative* can assume that *input* and *output* are ndarrays. Note that the output from *derivative* is modified inplace; be careful to copy important inputs before returning them.

output : array, optional

The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional

The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

extra_keywords : dict, optional

dict of extra keyword arguments to pass to passed function

extra_arguments : sequence, optional

Sequence of extra positional arguments to pass to passed function

`scipy.ndimage.generic_laplace`(*input*, *derivative2*, *output=None*, *mode='reflect'*, *cval=0.0*, *extra_arguments=()*, *extra_keywords=None*)

N-dimensional Laplace filter using a provided second derivative function

Parameters **input** : array_like

Input array to filter.

derivative2 : callable

Callable with the following signature:

```
derivative2(input, axis, output, mode, cval,
            *extra_arguments, **extra_keywords)
```

See *extra_arguments*, *extra_keywords* below.

output : array, optional

The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional

The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

extra_keywords : dict, optional
dict of extra keyword arguments to pass to passed function

extra_arguments : sequence, optional
Sequence of extra positional arguments to pass to passed function

`scipy.ndimage.laplace` (*input*, *output=None*, *mode='reflect'*, *cval=0.0*)
N-dimensional Laplace filter based on approximate second derivatives.

Parameters

input : array_like
Input array to filter.

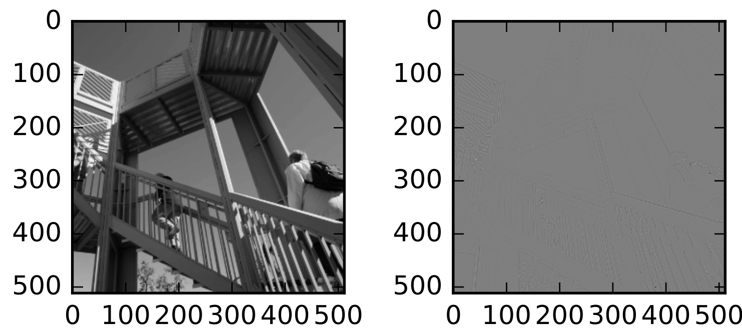
output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.laplace(ascent)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



scipy.ndimage.**maximum_filter**(*input*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculates a multi-dimensional maximum filter.

Parameters **input** : array_like
Input array to filter.

size : scalar or tuple, optional
See footprint, below

footprint : array, optional
Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

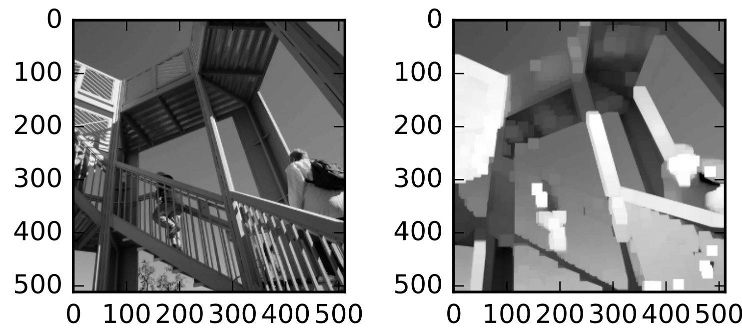
cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Returns **maximum_filter** : ndarray
Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.maximum_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.`**`maximum_filter1d`**(*input*, *size*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculate a one-dimensional maximum filter along the given axis.

The lines of the array along the given axis are filtered with a maximum filter of given size.

Parameters **input** : array_like
Input array to filter.
size : int
Length along which to calculate the 1-D maximum.
axis : int, optional
The axis of *input* along which to calculate. Default is -1.
output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.
mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'
cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0
origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Returns **maximum1d** : ndarray, None
Maximum-filtered array with same shape as input. None if *output* is not None

Notes

This function implements the MAXLIST algorithm *[R151]*, as described by Richard Harter *[R152]*, and has a guaranteed O(n) performance, *n* being the *input* length, regardless of filter size.

References

[R151], *[R152]*

`scipy.ndimage.`**`median_filter`**(*input*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculates a multidimensional median filter.

Parameters **input** : array_like

Input array to filter.

size : scalar or tuple, optional
See footprint, below

footprint : array, optional
Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

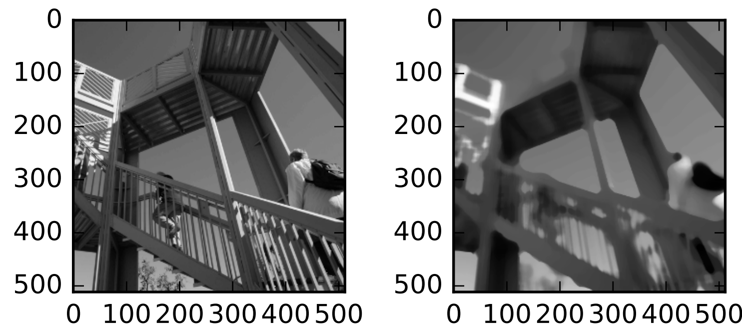
cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Returns **median_filter** : ndarray
Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.median_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.minimum_filter`(*input*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculates a multi-dimensional minimum filter.

Parameters

input : array_like
: Input array to filter.

size : scalar or tuple, optional
: See footprint, below

footprint : array, optional
: Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional
: The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
: The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
: Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
: The *origin* parameter controls the placement of the filter. Default 0.0.

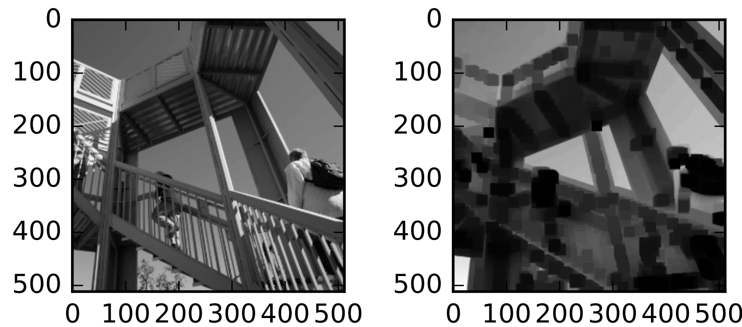
Returns

minimum_filter : ndarray
: Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
```

```
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.minimum_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.minimum_filter1d`(*input*, *size*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculate a one-dimensional minimum filter along the given axis.

The lines of the array along the given axis are filtered with a minimum filter of given size.

Parameters **input** : array_like
Input array to filter.

size : int
length along which to calculate 1D minimum

axis : int, optional
The axis of *input* along which to calculate. Default is -1.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Notes

This function implements the MINLIST algorithm *[R153]*, as described by Richard Harter *[R154]*, and has a guaranteed O(n) performance, *n* being the *input* length, regardless of filter size.

References

[R153], [R154]

`scipy.ndimage.`**`percentile_filter`**(*input*, *percentile*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculates a multi-dimensional percentile filter.

Parameters

input : array_like
    Input array to filter.

percentile : scalar
    The percentile parameter may be less then zero, i.e., percentile = -20 equals percentile = 80

size : scalar or tuple, optional
    See footprint, below

footprint : array, optional
    Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional
    The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
    The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
    Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

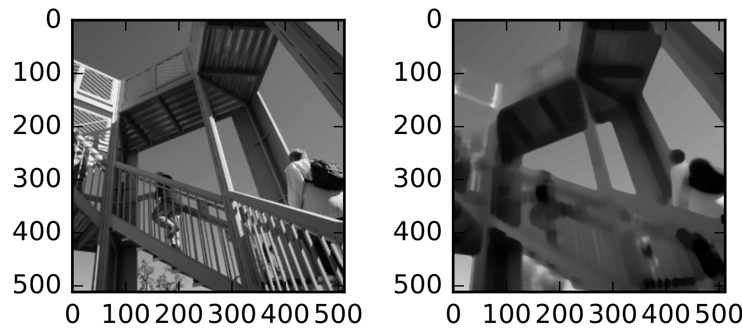
origin : scalar, optional
    The *origin* parameter controls the placement of the filter. Default 0.0.

Returns

percentile_filter : ndarray
    Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.percentile_filter(ascent, percentile=20, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.prewitt`(*input*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*)

Calculate a Prewitt filter.

Parameters

input : array_like
: Input array to filter.

axis : int, optional
: The axis of *input* along which to calculate. Default is -1.

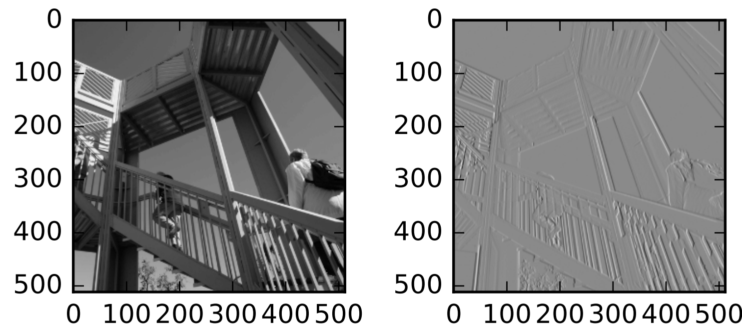
output : array, optional
: The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
: The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
: Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.prewitt(ascent)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.rank_filter`(*input*, *rank*, *size=None*, *footprint=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculates a multi-dimensional rank filter.

Parameters **input** : array_like

Input array to filter.

rank : int

The rank parameter may be less then zero, i.e., rank = -1 indicates the largest element.

size : scalar or tuple, optional

See footprint, below

footprint : array, optional

Either *size* or *footprint* must be defined. *size* gives the shape that is taken from the input array, at every element position, to define the input to the filter function. *footprint* is a boolean array that specifies (implicitly) a shape, but also which of the elements within this shape will get passed to the filter function. Thus `size=(n,m)` is equivalent to `footprint=np.ones((n,m))`. We adjust *size* to the number of dimensions of the input array, so that, if the input array is shape (10,10,10), and *size* is 2, then the actual size used is (2,2,2).

output : array, optional

The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional

The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional

The *origin* parameter controls the placement of the filter. Default 0.0.

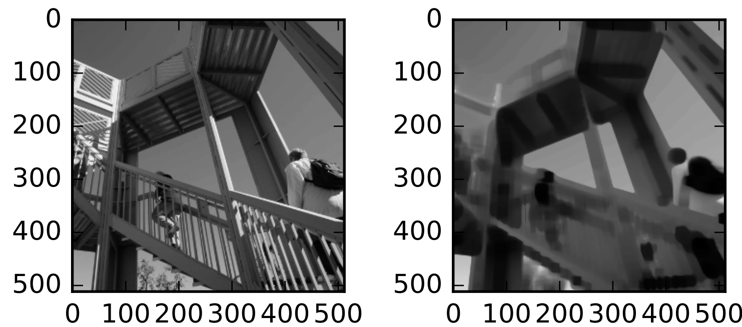
Returns **rank_filter** : ndarray

Filtered array. Has the same shape as *input*.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
```

```
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.rank_filter(ascent, rank=42, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.sobel`(*input*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*)

Calculate a Sobel filter.

Parameters

input : array_like
Input array to filter.

axis : int, optional
The axis of *input* along which to calculate. Default is -1.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

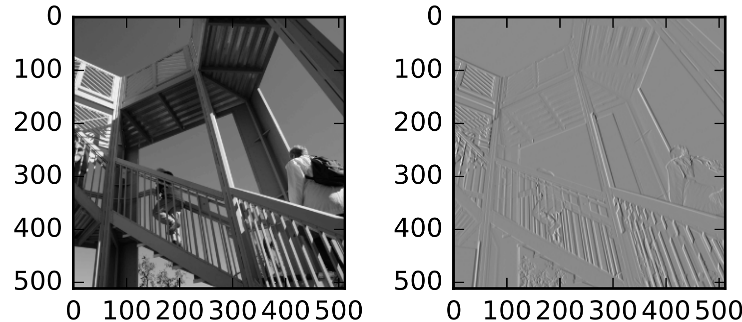
mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.sobel(ascent)
```

```
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.uniform_filter`(*input*, *size=3*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional uniform filter.

Parameters

input : array_like
Input array to filter.

size : int or sequence of ints, optional
The sizes of the uniform filter are given for each axis as a sequence, or as a single number, in which case the size is equal for all axes.

output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.

mode : str or sequence, optional
The *mode* parameter determines how the array borders are handled. Valid modes are {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}. *cval* is the value used when mode is equal to 'constant'. A list of modes with length equal to the number of axes can be provided to specify different modes for different axes. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

Returns

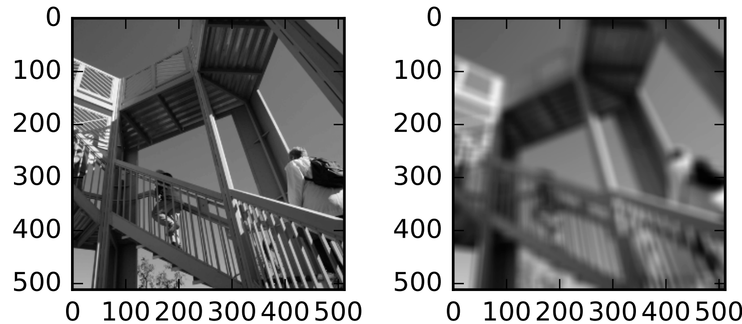
uniform_filter : ndarray
Filtered array. Has the same shape as *input*.

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.gray()  # show the filtered result in grayscale
>>> ax1 = fig.add_subplot(121)  # left side
>>> ax2 = fig.add_subplot(122)  # right side
>>> ascent = misc.ascent()
>>> result = ndimage.uniform_filter(ascent, size=20)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result)
>>> plt.show()
```



`scipy.ndimage.`**`uniform_filter1d`**(*input*, *size*, *axis=-1*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculate a one-dimensional uniform filter along the given axis.

The lines of the array along the given axis are filtered with a uniform filter of given size.

Parameters **input** : array_like
Input array to filter.
size : int
length of uniform filter
axis : int, optional
The axis of *input* along which to calculate. Default is -1.
output : array, optional
The *output* parameter passes an array in which to store the filter output. Output array should have different name as compared to input array to avoid aliasing errors.
mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'
cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0
origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0.0.

5.16.2 Fourier filters

`fourier_ellipsoid`(input, size[, n, axis, output])	Multi-dimensional ellipsoid fourier filter.
`fourier_gaussian`(input, sigma[, n, axis, output])	Multi-dimensional Gaussian fourier filter.
`fourier_shift`(input, shift[, n, axis, output])	Multi-dimensional fourier shift filter.
`fourier_uniform`(input, size[, n, axis, output])	Multi-dimensional uniform fourier filter.

scipy.ndimage.**fourier_ellipsoid**(*input*, *size*, *n=-1*, *axis=-1*, *output=None*)

Multi-dimensional ellipsoid fourier filter.

The array is multiplied with the fourier transform of a ellipsoid of given sizes.

Parameters

input : array_like
: The input array.

size : float or sequence
: The size of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.

n : int, optional
: If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.

axis : int, optional
: The axis of the real transform.

output : ndarray, optional
: If given, the result of filtering the input is placed in this array. None is returned in this case.

Returns

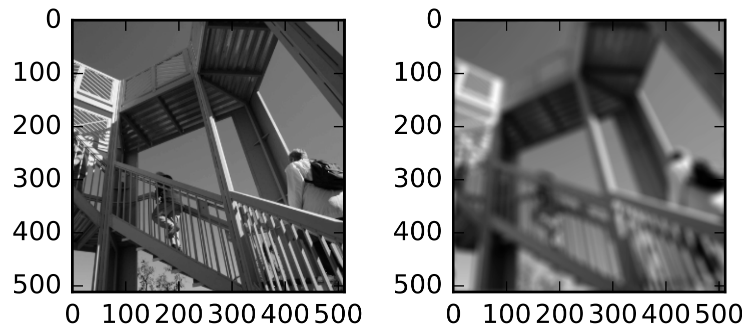
fourier_ellipsoid : ndarray or None
: The filtered input. If *output* is given as a parameter, None is returned.

Notes

This function is implemented for arrays of rank 1, 2, or 3.

Examples

```
>>> from scipy import ndimage, misc
>>> import numpy.fft
>>> import matplotlib.pyplot as plt
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_ellipsoid(input_, size=20)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```



scipy.ndimage.**fourier_gaussian**(*input*, *sigma*, *n=-1*, *axis=-1*, *output=None*)

Multi-dimensional Gaussian fourier filter.

The array is multiplied with the fourier transform of a Gaussian kernel.

Parameters

input : array_like
: The input array.

sigma : float or sequence
: The sigma of the Gaussian kernel. If a float, *sigma* is the same for all axes. If a sequence, *sigma* has to contain one value for each axis.

n : int, optional
: If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.

axis : int, optional
: The axis of the real transform.

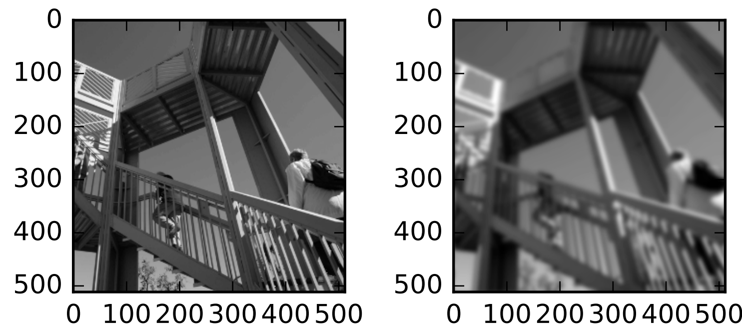
output : ndarray, optional
: If given, the result of filtering the input is placed in this array. None is returned in this case.

Returns

fourier_gaussian : ndarray or None
: The filtered input. If *output* is given as a parameter, None is returned.

Examples

```
>>> from scipy import ndimage, misc
>>> import numpy.fft
>>> import matplotlib.pyplot as plt
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_gaussian(input_, sigma=4)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```



`scipy.ndimage.fourier_shift`(*input*, *shift*, *n=-1*, *axis=-1*, *output=None*)

Multi-dimensional fourier shift filter.

The array is multiplied with the fourier transform of a shift operation.

Parameters **input** : array_like

The input array.

shift : float or sequence

The size of the box used for filtering. If a float, `shift` is the same for all axes. If a sequence, `shift` has to contain one value for each axis.

n : int, optional

If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.

axis : int, optional

The axis of the real transform.

output : ndarray, optional

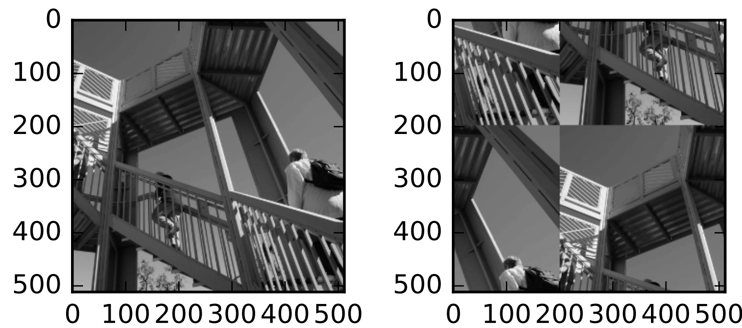
If given, the result of shifting the input is placed in this array. None is returned in this case.

Returns **fourier_shift** : ndarray or None

The shifted input. If *output* is given as a parameter, None is returned.

Examples

```
>>> from scipy import ndimage, misc
>>> import matplotlib.pyplot as plt
>>> import numpy.fft
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_shift(input_, shift=200)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```



scipy.ndimage.**fourier_uniform**(*input*, *size*, *n=-1*, *axis=-1*, *output=None*)

Multi-dimensional uniform fourier filter.

The array is multiplied with the fourier transform of a box of given size.

Parameters **input** : array_like

The input array.

size : float or sequence

The size of the box used for filtering. If a float, *size* is the same for all axes. If a sequence, *size* has to contain one value for each axis.

n : int, optional

If *n* is negative (default), then the input is assumed to be the result of a complex fft. If *n* is larger than or equal to zero, the input is assumed to be the result of a real fft, and *n* gives the length of the array before transformation along the real transform direction.

axis : int, optional

The axis of the real transform.

output : ndarray, optional

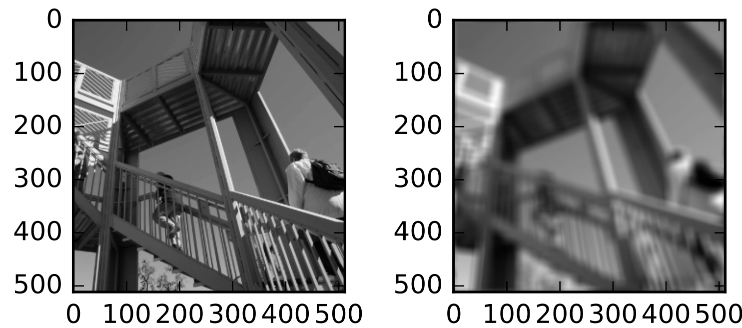
If given, the result of filtering the input is placed in this array. None is returned in this case.

Returns **fourier_uniform** : ndarray or None

The filtered input. If *output* is given as a parameter, None is returned.

Examples

```
>>> from scipy import ndimage, misc
>>> import numpy.fft
>>> import matplotlib.pyplot as plt
>>> fig, (ax1, ax2) = plt.subplots(1, 2)
>>> plt.gray()  # show the filtered result in grayscale
>>> ascent = misc.ascent()
>>> input_ = numpy.fft.fft2(ascent)
>>> result = ndimage.fourier_uniform(input_, size=20)
>>> result = numpy.fft.ifft2(result)
>>> ax1.imshow(ascent)
>>> ax2.imshow(result.real)  # the imaginary part is an artifact
>>> plt.show()
```



5.16.3 Interpolation

affine_transform(input, matrix[, offset, ...])	Apply an affine transformation.
geometric_transform(input, mapping[, ...])	Apply an arbitrary geometric transform.
map_coordinates(input, coordinates[, ...])	Map the input array to new coordinates by interpolation.
rotate(input, angle[, axes, reshape, ...])	Rotate an array.
shift(input, shift[, output, order, mode, ...])	Shift an array.
spline_filter(input[, order, output])	Multi-dimensional spline filter.
spline_filter1d(input[, order, axis, output])	Calculates a one-dimensional spline filter along the given axis.
zoom(input, zoom[, output, order, mode, ...])	Zoom an array.

`scipy.ndimage.`**`affine_transform`**(*input*, *matrix*, *offset=0.0*, *output_shape=None*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)

Apply an affine transformation.

The given matrix and offset are used to find for each point in the output the corresponding coordinates in the input by an affine transformation. The value of the input at those coordinates is determined by spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode.

Given an output image pixel index vector `o`, the pixel value is determined from the input image at position `np.dot(matrix,o) + offset`.

A diagonal matrix can be specified by supplying a one-dimensional array-like to the matrix parameter, in which case a more efficient algorithm is applied.

Changed in version 0.18.0: Previously, the exact interpretation of the affine transformation depended on whether the matrix was supplied as a one-dimensional or two-dimensional array. If a one-dimensional array was supplied to the matrix parameter, the output pixel value at index `o` was determined from the input image at position `matrix * (o + offset)`.

Parameters **input** : ndarray

The input array.

matrix : ndarray

The matrix must be two-dimensional or can also be given as a one-dimensional sequence or array. In the latter case, it is assumed that the matrix is diagonal. A more efficient algorithms is then applied that exploits the separability of the problem.

offset : float or sequence, optional
The offset into the array where the transform is applied. If a float, *offset* is the same for each axis. If a sequence, *offset* should contain one value for each axis.

output_shape : tuple of ints, optional
Shape tuple.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *spline_filter* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

Returns **affine_transform** : ndarray or None
The transformed input. If *output* is given as a parameter, None is returned.

`scipy.ndimage.`**`geometric_transform`**(*input*, *mapping*, *output_shape=None*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*, *extra_arguments=()*, *extra_keywords={}*)

Apply an arbitrary geometric transform.

The given mapping function is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

Parameters **input** : array_like
The input array.

mapping : {callable, scipy.LowLevelCallable}
A callable object that accepts a tuple of length equal to the output array rank, and returns the corresponding input coordinates as a tuple of length equal to the input array rank.

output_shape : tuple of ints, optional
Shape tuple.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *spline_filter* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

extra_arguments : tuple, optional

Extra arguments passed to *mapping*.

extra_keywords : dict, optional
Extra keywords passed to *mapping*.

Returns

return_value : ndarray or None
The filtered input. If *output* is given as a parameter, None is returned.

See also:

`map_coordinates`, `affine_transform`, `spline_filter1d`

Notes

This function also accepts low-level callback functions with one the following signatures and wrapped in `scipy.LowLevelCallable`:

```
int mapping(npy_intp *output_coordinates, double *input_coordinates,
            int output_rank, int input_rank, void *user_data)
int mapping(intptr_t *output_coordinates, double *input_coordinates,
            int output_rank, int input_rank, void *user_data)
```

The calling function iterates over the elements of the output array, calling the callback function at each element. The coordinates of the current output element are passed through `output_coordinates`. The callback function must return the coordinates at which the input must be interpolated in `input_coordinates`. The rank of the input and output arrays are given by `input_rank` and `output_rank` respectively. `user_data` is the data pointer provided to `scipy.LowLevelCallable` as-is.

The callback function must return an integer error status that is zero if something went wrong and one otherwise. If an error occurs, you should normally set the python error status with an informative message before returning, otherwise a default error message is set by the calling function.

In addition, some other low-level function pointer specifications are accepted, but these are for backward compatibility only and should not be used in new code.

Examples

```
>>> from scipy import ndimage
>>> a = np.arange(12.).reshape((4, 3))
>>> def shift_func(output_coords):
...     return (output_coords[0] - 0.5, output_coords[1] - 0.5)
...
>>> ndimage.geometric_transform(a, shift_func)
array([[ 0.   ,  0.   ,  0.   ],
       [ 0.   ,  1.362,  2.738],
       [ 0.   ,  4.812,  6.187],
       [ 0.   ,  8.263,  9.637]])
```

`scipy.ndimage.`**`map_coordinates`**(*input*, *coordinates*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)

Map the input array to new coordinates by interpolation.

The array of coordinates is used to find, for each point in the output, the corresponding coordinates in the input. The value of the input at those coordinates is determined by spline interpolation of the requested order.

The shape of the output is derived from that of the coordinate array by dropping the first axis. The values of the array along the first axis are the coordinates in the input array at which the output value is found.

Parameters

input : ndarray
The input array.

coordinates : array_like
The coordinates at which *input* is evaluated.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *spline_filter* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

Returns **map_coordinates** : ndarray
The result of transforming the input. The shape of the output is derived from that of *coordinates* by dropping the first axis.

See also:

spline_filter, *geometric_transform*, *scipy.interpolate*

Examples

```
>>> from scipy import ndimage
>>> a = np.arange(12.).reshape((4, 3))
>>> a
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
>>> ndimage.map_coordinates(a, [[0.5, 2], [0.5, 1]], order=1)
array([ 2.,  7.])
```

Above, the interpolated value of a[0.5, 0.5] gives output[0], while a[2, 1] is output[1].

```
>>> inds = np.array([[0.5, 2], [0.5, 4]])
>>> ndimage.map_coordinates(a, inds, order=1, cval=-33.3)
array([  2. , -33.3])
>>> ndimage.map_coordinates(a, inds, order=1, mode='nearest')
array([ 2.,  8.])
>>> ndimage.map_coordinates(a, inds, order=1, cval=0, output=bool)
array([ True, False], dtype=bool)
```

`scipy.ndimage.`**`rotate`**(*input*, *angle*, *axes=(1, 0)*, *reshape=True*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)

Rotate an array.

The array is rotated in the plane defined by the two axes given by the *axes* parameter using spline interpolation of the requested order.

Parameters **input** : ndarray
The input array.

angle : float
The rotation angle in degrees.

axes : tuple of 2 ints, optional
The two axes that define the plane of rotation. Default is the first two axes.

reshape : bool, optional

If *reshape* is true, the output shape is adapted so that the input array is contained completely in the output. Default is True.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *spline_filter* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

Returns **rotate** : ndarray or None
The rotated input. If *output* is given as a parameter, None is returned.

`scipy.ndimage.`**`shift`**(*input*, *shift*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)

Shift an array.

The array is shifted using spline interpolation of the requested order. Points outside the boundaries of the input are filled according to the given mode.

Parameters **input** : ndarray
The input array.

shift : float or sequence, optional
The shift along the axes. If a float, *shift* is the same for each axis. If a sequence, *shift* should contain one value for each axis.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *spline_filter* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

Returns **shift** : ndarray or None
The shifted input. If *output* is given as a parameter, None is returned.

`scipy.ndimage.`**`spline_filter`**(*input*, *order=3*, *output=<type 'numpy.float64'>*)

Multi-dimensional spline filter.

For more details, see *spline_filter1d*.

See also:

spline_filter1d

Notes

The multi-dimensional filter is implemented as a sequence of one-dimensional spline filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

`scipy.ndimage.`**`spline_filter1d`**(*input*, *order=3*, *axis=-1*, *output=<type 'numpy.float64'>*)

Calculates a one-dimensional spline filter along the given axis.

The lines of the array along the given axis are filtered by a spline filter. The order of the spline must be >= 2 and <= 5.

Parameters

input : array_like
The input array.

order : int, optional
The order of the spline, default is 3.

axis : int, optional
The axis along which the spline filter is applied. Default is the last axis.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array. Default is `numpy.float64`.

Returns

spline_filter1d : ndarray or None
The filtered input. If *output* is given as a parameter, None is returned.

`scipy.ndimage.`**`zoom`**(*input*, *zoom*, *output=None*, *order=3*, *mode='constant'*, *cval=0.0*, *prefilter=True*)

Zoom an array.

The array is zoomed using spline interpolation of the requested order.

Parameters

input : ndarray
The input array.

zoom : float or sequence, optional
The zoom factor along the axes. If a float, *`zoom`* is the same for each axis. If a sequence, *`zoom`* should contain one value for each axis.

output : ndarray or dtype, optional
The array in which to place the output, or the dtype of the returned array.

order : int, optional
The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

mode : str, optional
Points outside the boundaries of the input are filled according to the given mode ('constant', 'nearest', 'reflect', 'mirror' or 'wrap'). Default is 'constant'.

cval : scalar, optional
Value used for points outside the boundaries of the input if `mode='constant'`. Default is 0.0

prefilter : bool, optional
The parameter prefilter determines if the input is pre-filtered with *`spline_filter`* before interpolation (necessary for spline interpolation of order > 1). If False, it is assumed that the input is already filtered. Default is True.

Returns

zoom : ndarray or None
The zoomed input. If *output* is given as a parameter, None is returned.

5.16.4 Measurements

`center_of_mass`(input[, labels, index])	Calculate the center of mass of the values of an array at labels.

Continued on next page

Table 5.92 – continued from previous page

extrema(input[, labels, index])	Calculate the minimums and maximums of the values of an array at labels, along with their positions.
find_objects(input[, max_label])	Find objects in a labeled array.
histogram(input, min, max, bins[, labels, index])	Calculate the histogram of the values of an array, optionally at labels.
label(input[, structure, output])	Label features in an array.
labeled_comprehension(input, labels, index, ...)	Roughly equivalent to [func(input[labels == i]) for i in index].
maximum(input[, labels, index])	Calculate the maximum of the values of an array over labeled regions.
maximum_position(input[, labels, index])	Find the positions of the maximums of the values of an array at labels.
mean(input[, labels, index])	Calculate the mean of the values of an array at labels.
median(input[, labels, index])	Calculate the median of the values of an array over labeled regions.
minimum(input[, labels, index])	Calculate the minimum of the values of an array over labeled regions.
minimum_position(input[, labels, index])	Find the positions of the minimums of the values of an array at labels.
standard_deviation(input[, labels, index])	Calculate the standard deviation of the values of an n-D image array, optionally at specified sub-regions.
sum(input[, labels, index])	Calculate the sum of the values of the array.
variance(input[, labels, index])	Calculate the variance of the values of an n-D image array, optionally at specified sub-regions.
watershed_ift(input, markers[, structure, ...])	Apply watershed from markers using image foresting transform algorithm.

scipy.ndimage.**center_of_mass**(*input*, *labels=None*, *index=None*)

Calculate the center of mass of the values of an array at labels.

Parameters **input** : ndarray

Data from which to calculate center-of-mass. The masses can either be positive or negative.

labels : ndarray, optional

Labels for objects in *input*, as generated by *ndimage.label*. Only used with *index*. Dimensions must be the same as *input*.

index : int or sequence of ints, optional

Labels for which to calculate centers-of-mass. If not specified, all labels greater than zero are used. Only used with *labels*.

Returns **center_of_mass** : tuple, or list of tuples

Coordinates of centers-of-mass.

Examples

```
>>> a = np.array(([0,0,0,0],
...               [0,1,1,0],
...               [0,1,1,0],
...               [0,1,1,0]))
>>> from scipy import ndimage
>>> ndimage.measurements.center_of_mass(a)
(2.0, 1.5)
```

Calculation of multiple objects in an image

```
>>> b = np.array(([0,1,1,0],
...               [0,1,0,0],
...               [0,0,0,0],
...               [0,0,1,1],
...               [0,0,1,1]))
>>> lbl = ndimage.label(b)[0]
>>> ndimage.measurements.center_of_mass(b, lbl, [1,2])
[(0.33333333333333331, 1.3333333333333333), (3.5, 2.5)]
```

Negative masses are also accepted, which can occur for example when bias is removed from measured data due to random noise.

```
>>> c = np.array(([-1,0,0,0],
...               [0,-1,-1,0],
...               [0,1,-1,0],
...               [0,1,1,0]))
>>> ndimage.measurements.center_of_mass(c)
(-4.0, 1.0)
```

If there are division by zero issues, the function does not raise an error but rather issues a RuntimeWarning before returning inf and/or NaN.

```
>>> d = np.array([-1, 1])
>>> ndimage.measurements.center_of_mass(d)
(inf,)
```

scipy.ndimage.**extrema**(*input*, *labels=None*, *index=None*)

Calculate the minimums and maximums of the values of an array at labels, along with their positions.

Parameters

input : ndarray
: Nd-image data to process.

labels : ndarray, optional
: Labels of features in input. If not None, must be same shape as *input*.

index : int or sequence of ints, optional
: Labels to include in output. If None (default), all values where non-zero *labels* are used.

Returns

minimums, maximums : int or ndarray
: Values of minimums and maximums in each feature.

min_positions, max_positions : tuple or list of tuples
: Each tuple gives the n-D coordinates of the corresponding minimum or maximum.

See also:

maximum, *minimum*, *maximum_position*, *minimum_position*, *center_of_mass*

Examples

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> from scipy import ndimage
>>> ndimage.extrema(a)
(0, 9, (0, 2), (3, 0))
```

Features to process can be specified using *labels* and *index*:

```
>>> lbl, nlbl = ndimage.label(a)
>>> ndimage.extrema(a, lbl, index=np.arange(1, nlbl+1))
(array([1, 4, 3]),
 array([5, 7, 9]),
 [(0, 0), (1, 3), (3, 1)],
 [(1, 0), (2, 3), (3, 0)])
```

If no index is given, non-zero *labels* are processed:

```
>>> ndimage.extrema(a, lbl)
(1, 9, (0, 0), (3, 0))
```

scipy.ndimage.**find_objects**(*input*, *max_label=0*)

Find objects in a labeled array.

Parameters

input : ndarray of ints
Array containing objects defined by different labels. Labels with value 0 are ignored.

max_label : int, optional
Maximum label to be searched for in *input*. If max_label is not given, the positions of all objects are returned.

Returns

object_slices : list of tuples
A list of tuples, with each tuple containing N slices (with N the dimension of the input array). Slices correspond to the minimal parallelepiped that contains the object. If a number is missing, None is returned instead of a slice.

See also:

label, *center_of_mass*

Notes

This function is very useful for isolating a volume of interest inside a 3-D array, that cannot be "seen through".

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((6,6), dtype=int)
>>> a[2:4, 2:4] = 1
>>> a[4, 4] = 1
>>> a[:2, :3] = 2
>>> a[0, 5] = 3
>>> a
array([[2, 2, 2, 0, 0, 3],
       [2, 2, 2, 0, 0, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 0, 1, 1, 0, 0],
       [0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0]])
>>> ndimage.find_objects(a)
[(slice(2, 5, None), slice(2, 5, None)), (slice(0, 2, None), slice(0, 3, None)),
↪(slice(0, 1, None), slice(5, 6, None))]
>>> ndimage.find_objects(a, max_label=2)
[(slice(2, 5, None), slice(2, 5, None)), (slice(0, 2, None), slice(0, 3, None))]
>>> ndimage.find_objects(a == 1, max_label=2)
[(slice(2, 5, None), slice(2, 5, None)), None]
```

```
>>> loc = ndimage.find_objects(a)[0]
>>> a[loc]
```

```
array([[1, 1, 0],
       [1, 1, 0],
       [0, 0, 1]])
```

`scipy.ndimage.histogram`(*input*, *min*, *max*, *bins*, *labels=None*, *index=None*)

Calculate the histogram of the values of an array, optionally at labels.

Histogram calculates the frequency of values in an array within bins determined by *min*, *max*, and *bins*. The *labels* and *index* keywords can limit the scope of the histogram to specified sub-regions within the array.

Parameters

input : array_like
Data for which to calculate histogram.

min, max : int
Minimum and maximum values of range of histogram bins.

bins : int
Number of bins.

labels : array_like, optional
Labels for objects in *input*. If not None, must be same shape as *input*.

index : int or sequence of ints, optional
Label or labels for which to calculate histogram. If None, all values where label is greater than zero are used

Returns

hist : ndarray
Histogram counts.

Examples

```
>>> a = np.array([[ 0.    ,  0.2146,  0.5962,  0.    ],
...               [ 0.    ,  0.7778,  0.    ,  0.    ],
...               [ 0.    ,  0.    ,  0.    ,  0.    ],
...               [ 0.    ,  0.    ,  0.7181,  0.2787],
...               [ 0.    ,  0.    ,  0.6573,  0.3094]])
>>> from scipy import ndimage
>>> ndimage.measurements.histogram(a, 0, 1, 10)
array([13,  0,  2,  1,  0,  1,  1,  2,  0,  0])
```

With labels and no indices, non-zero elements are counted:

```
>>> lbl, nlbl = ndimage.label(a)
>>> ndimage.measurements.histogram(a, 0, 1, 10, lbl)
array([0, 0, 2, 1, 0, 1, 1, 2, 0, 0])
```

Indices can be used to count only certain objects:

```
>>> ndimage.measurements.histogram(a, 0, 1, 10, lbl, 2)
array([0, 0, 1, 1, 0, 0, 1, 1, 0, 0])
```

`scipy.ndimage.label`(*input*, *structure=None*, *output=None*)

Label features in an array.

Parameters

input : array_like
An array-like object to be labeled. Any non-zero values in *input* are counted as features and zero values are considered the background.

structure : array_like, optional
A structuring element that defines feature connections. *structure* must be symmetric. If no structuring element is provided, one is automatically generated with a squared connectivity equal to one. That is, for a 2-D *input* array, the default structuring element is:

```
[[0,1,0],
 [1,1,1],
 [0,1,0]]
```

output : (None, data-type, array_like), optional
If *output* is a data type, it specifies the type of the resulting labeled feature array If *output* is an array-like object, then *output* will be updated with the labeled features from this function. This function can operate in-place, by passing output=input. Note that the output must be able to store the largest label, or this function will raise an Exception.

Returns **label** : ndarray or int
An integer ndarray where each unique feature in *input* has a unique label in the returned array.

num_features : int
How many objects were found.
If *output* is None, this function returns a tuple of (*labeled_array*, *num_features*).
If *output* is a ndarray, then it will be updated with values in *labeled_array* and only *num_features* will be returned by this function.

See also:

find_objects
generate a list of slices for the labeled features (or objects); useful for finding features' position or dimensions

Examples

Create an image with some features, then label it using the default (cross-shaped) structuring element:

```
>>> from scipy.ndimage import label, generate_binary_structure
>>> a = np.array([[0,0,1,1,0,0],
...               [0,0,0,1,0,0],
...               [1,1,0,0,1,0],
...               [0,0,0,1,0,0]])
>>> labeled_array, num_features = label(a)
```

Each of the 4 features are labeled with a different integer:

```
>>> num_features
4
>>> labeled_array
array([[0, 0, 1, 1, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [2, 2, 0, 0, 3, 0],
       [0, 0, 0, 4, 0, 0]])
```

Generate a structuring element that will consider features connected even if they touch diagonally:

```
>>> s = generate_binary_structure(2,2)
```

or,

```
>>> s = [[1,1,1],
...      [1,1,1],
...      [1,1,1]]
```

Label the image using the new structuring element:

```
>>> labeled_array, num_features = label(a, structure=s)
```

Show the 2 labeled features (note that features 1, 3, and 4 from above are now considered a single feature):

```
>>> num_features
2
>>> labeled_array
array([[0, 0, 1, 1, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [2, 2, 0, 0, 1, 0],
       [0, 0, 0, 1, 0, 0]])
```

scipy.ndimage.**labeled_comprehension**(*input*, *labels*, *index*, *func*, *out_dtype*, *default*, *pass_positions=False*)

Roughly equivalent to [func(input[labels == i]) for i in index].

Sequentially applies an arbitrary function (that works on array_like input) to subsets of an n-D image array specified by *labels* and *index*. The option exists to provide the function with positional parameters as the second argument.

Parameters

input : array_like
: Data from which to select *labels* to process.

labels : array_like or None
: Labels to objects in *input*. If not None, array must be same shape as *input*. If None, *func* is applied to raveled *input*.

index : int, sequence of ints or None
: Subset of *labels* to which to apply *func*. If a scalar, a single value is returned. If None, *func* is applied to all non-zero values of *labels*.

func : callable
: Python function to apply to *labels* from *input*.

out_dtype : dtype
: Dtype to use for *result*.

default : int, float or None
: Default return value when a element of *index* does not exist in *labels*.

pass_positions : bool, optional
: If True, pass linear indices to *func* as a second argument. Default is False.

Returns

result : ndarray
: Result of applying *func* to each of *labels* to *input* in *index*.

Examples

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> from scipy import ndimage
>>> lbl, nlbl = ndimage.label(a)
>>> lbls = np.arange(1, nlbl+1)
>>> ndimage.labeled_comprehension(a, lbl, lbls, np.mean, float, 0)
array([ 2.75,  5.5 ,  6.  ])
```

Falling back to *default*:

```
>>> lbls = np.arange(1, nlbl+2)
>>> ndimage.labeled_comprehension(a, lbl, lbls, np.mean, float, -1)
array([ 2.75,  5.5 ,  6.  , -1.  ])
```

Passing positions:

```
>>> def fn(val, pos):
...     print("fn says: %s : %s" % (val, pos))
...     return (val.sum()) if (pos.sum() % 2 == 0) else (-val.sum())
...
>>> ndimage.labeled_comprehension(a, lbl, lbls, fn, float, 0, True)
fn says: [1 2 5 3] : [0 1 4 5]
fn says: [4 7] : [ 7 11]
fn says: [9 3] : [12 13]
array([ 11.,  11., -12.,   0.])
```

scipy.ndimage.**maximum**(*input*, *labels=None*, *index=None*)

Calculate the maximum of the values of an array over labeled regions.

Parameters

input : array_like
: Array_like of values. For each region specified by *labels*, the maximal values of *input* over the region is computed.

labels : array_like, optional
: An array of integers marking different regions over which the maximum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the maximum over the whole array is returned.

index : array_like, optional
: A list of region labels that are taken into account for computing the maxima. If index is None, the maximum over all elements where *labels* is non-zero is returned.

Returns

output : float or list of floats
: List of maxima of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a float is returned: the maximal value of *input* if *labels* is None, and the maximal value of elements where *labels* is greater than zero if *index* is None.

See also:

label, *minimum*, *median*, *maximum_position*, *extrema*, *sum*, *mean*, *variance*, *standard_deviation*

Notes

The function returns a Python list and not a Numpy array, use *np.array* to convert the list to an array.

Examples

```
>>> a = np.arange(16).reshape((4,4))
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> labels = np.zeros_like(a)
>>> labels[:2,:2] = 1
>>> labels[2:, 1:3] = 2
>>> labels
array([[1, 1, 0, 0],
       [1, 1, 0, 0],
       [0, 2, 2, 0],
       [0, 2, 2, 0]])
>>> from scipy import ndimage
>>> ndimage.maximum(a)
```

```
15.0
>>> ndimage.maximum(a, labels=labels, index=[1,2])
[5.0, 14.0]
>>> ndimage.maximum(a, labels=labels)
14.0
```

```
>>> b = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> labels, labels_nb = ndimage.label(b)
>>> labels
array([[1, 1, 0, 0],
       [1, 1, 0, 2],
       [0, 0, 0, 2],
       [3, 3, 0, 0]])
>>> ndimage.maximum(b, labels=labels, index=np.arange(1, labels_nb + 1))
[5.0, 7.0, 9.0]
```

`scipy.ndimage.`**`maximum_position`**(*input*, *labels=None*, *index=None*)

Find the positions of the maximums of the values of an array at labels.

For each region specified by *labels*, the position of the maximum value of *input* within the region is returned.

Parameters

input : array_like
Array_like of values.

labels : array_like, optional
An array of integers marking different regions over which the position of the maximum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first maximum over the whole array is returned.
The *labels* argument only works when *index* is specified.

index : array_like, optional
A list of region labels that are taken into account for finding the location of the maxima. If *index* is None, the first maximum over all elements where *labels* is non-zero is returned.
The *index* argument only works when *labels* is specified.

Returns

output : list of tuples of ints
List of tuples of ints that specify the location of maxima of *input* over the regions determined by *labels* and whose index is in *index*.
If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the `first` maximal value of *input*.

See also:

label, *minimum*, *median*, *maximum_position*, *extrema*, *sum*, *mean*, *variance*, *standard_deviation*

`scipy.ndimage.`**`mean`**(*input*, *labels=None*, *index=None*)

Calculate the mean of the values of an array at labels.

Parameters

input : array_like
Array on which to compute the mean of elements over distinct regions.

labels : array_like, optional
Array of labels of same shape, or broadcastable to the same shape as *input*. All elements sharing the same label form one region over which the mean of the elements is computed.

index : int or sequence of ints, optional

Labels of the objects over which the mean is to be computed. Default is None, in which case the mean for all values where label is greater than 0 is calculated.

Returns **out** : list

Sequence of same length as *index*, with the mean of the different regions labeled by the labels in *index*.

See also:

`ndimage.variance`, `ndimage.standard_deviation`, `ndimage.minimum`, `ndimage.maximum`, `ndimage.sum`, `ndimage.label`

Examples

```
>>> from scipy import ndimage
>>> a = np.arange(25).reshape((5,5))
>>> labels = np.zeros_like(a)
>>> labels[3:5,3:5] = 1
>>> index = np.unique(labels)
>>> labels
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 0, 1, 1]])
>>> index
array([0, 1])
>>> ndimage.mean(a, labels=labels, index=index)
[10.285714285714286, 21.0]
```

`scipy.ndimage.`**`median`**(*input*, *labels=None*, *index=None*)

Calculate the median of the values of an array over labeled regions.

Parameters **input** : array_like

Array_like of values. For each region specified by *labels*, the median value of *input* over the region is computed.

labels : array_like, optional

An array_like of integers marking different regions over which the median value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the median over the whole array is returned.

index : array_like, optional

A list of region labels that are taken into account for computing the medians. If index is None, the median over all elements where *labels* is non-zero is returned.

Returns **median** : float or list of floats

List of medians of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a float is returned: the median value of *input* if *labels* is None, and the median value of elements where *labels* is greater than zero if *index* is None.

See also:

`label`, *`minimum`*, *`maximum`*, *`extrema`*, *`sum`*, *`mean`*, *`variance`*, *`standard_deviation`*

Notes

The function returns a Python list and not a Numpy array, use *np.array* to convert the list to an array.

Examples

```
>>> from scipy import ndimage
>>> a = np.array([[1, 2, 0, 1],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> labels, labels_nb = ndimage.label(a)
>>> labels
array([[1, 1, 0, 2],
       [1, 1, 0, 2],
       [0, 0, 0, 2],
       [3, 3, 0, 0]])
>>> ndimage.median(a, labels=labels, index=np.arange(1, labels_nb + 1))
[2.5, 4.0, 6.0]
>>> ndimage.median(a)
1.0
>>> ndimage.median(a, labels=labels)
3.0
```

`scipy.ndimage.`**`minimum`**`(`*input*, *labels=None*, *index=None*`)`

Calculate the minimum of the values of an array over labeled regions.

Parameters

input : array_like
: Array_like of values. For each region specified by *labels*, the minimal values of *input* over the region is computed.

labels : array_like, optional
: An array_like of integers marking different regions over which the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the minimum over the whole array is returned.

index : array_like, optional
: A list of region labels that are taken into account for computing the minima. If index is None, the minimum over all elements where *labels* is non-zero is returned.

Returns

minimum : float or list of floats
: List of minima of *input* over the regions determined by *labels* and whose index is in *index*. If *index* or *labels* are not specified, a float is returned: the minimal value of *input* if *labels* is None, and the minimal value of elements where *labels* is greater than zero if *index* is None.

See also:

label, *maximum*, *median*, *minimum_position*, *extrema*, *sum*, *mean*, *variance*, *standard_deviation*

Notes

The function returns a Python list and not a Numpy array, use *np.array* to convert the list to an array.

Examples

```
>>> from scipy import ndimage
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> labels, labels_nb = ndimage.label(a)
>>> labels
array([[1, 1, 0, 0],
```

```
       [1, 1, 0, 2],
       [0, 0, 0, 2],
       [3, 3, 0, 0]])
>>> ndimage.minimum(a, labels=labels, index=np.arange(1, labels_nb + 1))
[1.0, 4.0, 3.0]
>>> ndimage.minimum(a)
0.0
>>> ndimage.minimum(a, labels=labels)
1.0
```

scipy.ndimage.**minimum_position**(*input*, *labels=None*, *index=None*)
Find the positions of the minimums of the values of an array at labels.

Parameters **input** : array_like
Array_like of values.
labels : array_like, optional
An array of integers marking different regions over which the position of the minimum value of *input* is to be computed. *labels* must have the same shape as *input*. If *labels* is not specified, the location of the first minimum over the whole array is returned.
The *labels* argument only works when *index* is specified.
index : array_like, optional
A list of region labels that are taken into account for finding the location of the minima. If *index* is None, the `first` minimum over all elements where *labels* is non-zero is returned.
The *index* argument only works when *labels* is specified.

Returns **output** : list of tuples of ints
Tuple of ints or list of tuples of ints that specify the location of minima of *input* over the regions determined by *labels* and whose index is in *index*.
If *index* or *labels* are not specified, a tuple of ints is returned specifying the location of the first minimal value of *input*.

See also:

label, *minimum*, *median*, *maximum_position*, *extrema*, *sum*, *mean*, *variance*, *standard_deviation*

scipy.ndimage.**standard_deviation**(*input*, *labels=None*, *index=None*)
Calculate the standard deviation of the values of an n-D image array, optionally at specified sub-regions.

Parameters **input** : array_like
Nd-image data to process.
labels : array_like, optional
Labels to identify sub-regions in *input*. If not None, must be same shape as *input*.
index : int or sequence of ints, optional
labels to include in output. If None (default), all values where *labels* is non-zero are used.

Returns **standard_deviation** : float or ndarray
Values of standard deviation, for each sub-region if *labels* and *index* are specified.

See also:

label, *variance*, *maximum*, *minimum*, *extrema*

Examples

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
```

```
...                [9, 3, 0, 0]])
>>> from scipy import ndimage
>>> ndimage.standard_deviation(a)
2.7585095613392387
```

Features to process can be specified using *labels* and *index*:

```
>>> lbl, nlbl = ndimage.label(a)
>>> ndimage.standard_deviation(a, lbl, index=np.arange(1, nlbl+1))
array([ 1.479,  1.5  ,  3.   ])
```

If no index is given, non-zero *labels* are processed:

```
>>> ndimage.standard_deviation(a, lbl)
2.4874685927665499
```

scipy.ndimage.**sum**(*input*, *labels=None*, *index=None*)

Calculate the sum of the values of the array.

Parameters **input** : array_like

Values of *input* inside the regions defined by *labels* are summed together.

labels : array_like of ints, optional

Assign labels to the values of the array. Has to have the same shape as *input*.

index : array_like, optional

A single label number or a sequence of label numbers of the objects to be measured.

Returns **sum** : ndarray or scalar

An array of the sums of values of *input* inside the regions defined by *labels* with the same shape as *index*. If 'index' is None or scalar, a scalar is returned.

See also:

mean, *median*

Examples

```
>>> from scipy import ndimage
>>> input =  [0,1,2,3]
>>> labels = [1,1,2,2]
>>> ndimage.sum(input, labels, index=[1,2])
[1.0, 5.0]
>>> ndimage.sum(input, labels, index=1)
1
>>> ndimage.sum(input, labels)
6
```

scipy.ndimage.**variance**(*input*, *labels=None*, *index=None*)

Calculate the variance of the values of an n-D image array, optionally at specified sub-regions.

Parameters **input** : array_like

Nd-image data to process.

labels : array_like, optional

Labels defining sub-regions in *input*. If not None, must be same shape as *input*.

index : int or sequence of ints, optional

labels to include in output. If None (default), all values where *labels* is non-zero are used.

Returns **variance** : float or ndarray

Values of variance, for each sub-region if *labels* and *index* are specified.

See also:

label, *standard_deviation*, *maximum*, *minimum*, *extrema*

Examples

```
>>> a = np.array([[1, 2, 0, 0],
...               [5, 3, 0, 4],
...               [0, 0, 0, 7],
...               [9, 3, 0, 0]])
>>> from scipy import ndimage
>>> ndimage.variance(a)
7.609375
```

Features to process can be specified using *labels* and *index*:

```
>>> lbl, nlbl = ndimage.label(a)
>>> ndimage.variance(a, lbl, index=np.arange(1, nlbl+1))
array([ 2.1875,  2.25  ,  9.    ])
```

If no index is given, all non-zero *labels* are processed:

```
>>> ndimage.variance(a, lbl)
6.1875
```

scipy.ndimage.**watershed_ift**(*input*, *markers*, *structure=None*, *output=None*)

Apply watershed from markers using image foresting transform algorithm.

Parameters

input : array_like
: Input.

markers : array_like
: Markers are points within each watershed that form the beginning of the process. Negative markers are considered background markers which are processed after the other markers.

structure : structure element, optional
: A structuring element defining the connectivity of the object can be provided. If None, an element is generated with a squared connectivity equal to one.

output : ndarray, optional
: An output array can optionally be provided. The same shape as input.

Returns

watershed_ift : ndarray
: Output. Same shape as *input*.

References

[R156]

5.16.5 Morphology

binary_closing(input[, structure, ...])	Multi-dimensional binary closing with the given structuring element.
binary_dilation(input[, structure, ...])	Multi-dimensional binary dilation with the given structuring element.
binary_erosion(input[, structure, ...])	Multi-dimensional binary erosion with a given structuring element.

Continued on next page

Table 5.93 – continued from previous page

binary_fill_holes(input[, structure, ...])	Fill the holes in binary objects.
binary_hit_or_miss(input[, structure1, ...])	Multi-dimensional binary hit-or-miss transform.
binary_opening(input[, structure, ...])	Multi-dimensional binary opening with the given structuring element.
binary_propagation(input[, structure, mask, ...])	Multi-dimensional binary propagation with the given structuring element.
black_tophat(input[, size, footprint, ...])	Multi-dimensional black tophat filter.
distance_transform_bf(input[, metric, ...])	Distance transform function by a brute force algorithm.
distance_transform_cdt(input[, metric, ...])	Distance transform for chamfer type of transforms.
distance_transform_edt(input[, sampling, ...])	Exact euclidean distance transform.
generate_binary_structure(rank, connectivity)	Generate a binary structure for binary morphological operations.
grey_closing(input[, size, footprint, ...])	Multi-dimensional greyscale closing.
grey_dilation(input[, size, footprint, ...])	Calculate a greyscale dilation, using either a structuring element, or a footprint corresponding to a flat structuring element.
grey_erosion(input[, size, footprint, ...])	Calculate a greyscale erosion, using either a structuring element, or a footprint corresponding to a flat structuring element.
grey_opening(input[, size, footprint, ...])	Multi-dimensional greyscale opening.
iterate_structure(structure, iterations[, ...])	Iterate a structure by dilating it with itself.
morphological_gradient(input[, size, ...])	Multi-dimensional morphological gradient.
morphological_laplace(input[, size, ...])	Multi-dimensional morphological laplace.
white_tophat(input[, size, footprint, ...])	Multi-dimensional white tophat filter.

scipy.ndimage.**binary_closing**(*input*, *structure=None*, *iterations=1*, *output=None*, *origin=0*)

Multi-dimensional binary closing with the given structuring element.

The *closing* of an input image by a structuring element is the *erosion* of the *dilation* of the image by the structuring element.

Parameters

input : array_like
Binary array_like to be closed. Non-zero (True) elements form the subset to be closed.

structure : array_like, optional
Structuring element used for the closing. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one (i.e., only nearest neighbors are connected to the center, diagonally-connected elements are not considered neighbors).

iterations : {int, float}, optional
The dilation step of the closing, then the erosion step are each repeated *iterations* times (one, by default). If iterations is less than 1, each operations is repeated until the result does not change anymore.

output : ndarray, optional
Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin : int or tuple of ints, optional
Placement of the filter, by default 0.

Returns

binary_closing : ndarray of bools
Closing of the input by the structuring element.

See also:

grey_closing, *binary_opening*, *binary_dilation*, *binary_erosion*, *generate_binary_structure*

Notes

Closing [R133] is a mathematical morphology operation *[R134]* that consists in the succession of a dilation and an erosion of the input with the same structuring element. Closing therefore fills holes smaller than the structuring element.

Together with *opening* (`binary_opening`), closing can be used for noise removal.

References

[R133], *[R134]*

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:-1, 1:-1] = 1; a[2,2] = 0
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Closing removes small holes
>>> ndimage.binary_closing(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Closing is the erosion of the dilation of the input
>>> ndimage.binary_dilation(a).astype(int)
array([[0, 1, 1, 1, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 1, 1, 1, 0]])
>>> ndimage.binary_erosion(ndimage.binary_dilation(a)).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
```

```
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 2:5] = 1; a[1:3,3] = 0
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # In addition to removing holes, closing can also
>>> # coarsen boundaries with fine hollows.
>>> ndimage.binary_closing(a).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 0, 0],
```

```
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_closing(a, structure=np.ones((2,2))).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

`scipy.ndimage.`**`binary_dilation`**(*input*, *structure=None*, *iterations=1*, *mask=None*, *output=None*, *border_value=0*, *origin=0*, *brute_force=False*)

Multi-dimensional binary dilation with the given structuring element.

Parameters **input** : array_like
Binary array_like to be dilated. Non-zero (True) elements form the subset to be dilated.

structure : array_like, optional
Structuring element used for the dilation. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one.

iterations : {int, float}, optional
The dilation is repeated *iterations* times (one, by default). If iterations is less than 1, the dilation is repeated until the result does not change anymore.

mask : array_like, optional
If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.

output : ndarray, optional
Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin : int or tuple of ints, optional
Placement of the filter, by default 0.

border_value : int (cast to 0 or 1), optional
Value at the border in the output array.

Returns **binary_dilation** : ndarray of bools
Dilation of the input by the structuring element.

See also:

grey_dilation, *binary_erosion*, *binary_closing*, *binary_opening*, *generate_binary_structure*

Notes

Dilation *[R135]* is a mathematical morphology operation *[R136]* that uses a structuring element for expanding the shapes in an image. The binary dilation of an image by a structuring element is the locus of the points covered by the structuring element, when its center lies within the non-zero points of the image.

References

[R135], *[R136]*

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5, 5))
>>> a[2, 2] = 1
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a)
array([[False, False, False, False, False],
       [False, False,  True, False, False],
       [False,  True,  True,  True, False],
       [False, False,  True, False, False],
       [False, False, False, False, False]], dtype=bool)
>>> ndimage.binary_dilation(a).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> # 3x3 structuring element with connectivity 1, used by default
>>> struct1 = ndimage.generate_binary_structure(2, 1)
>>> struct1
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> # 3x3 structuring element with connectivity 2
>>> struct2 = ndimage.generate_binary_structure(2, 2)
>>> struct2
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
>>> ndimage.binary_dilation(a, structure=struct1).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a, structure=struct2).astype(a.dtype)
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(a, structure=struct1,\
... iterations=2).astype(a.dtype)
array([[ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.]])
```

`scipy.ndimage.`**`binary_erosion`**(*input*, *structure=None*, *iterations=1*, *mask=None*, *output=None*, *border_value=0*, *origin=0*, *brute_force=False*)

Multi-dimensional binary erosion with a given structuring element.

Binary erosion is a mathematical morphology operation used for image processing.

Parameters **input** : array_like
Binary image to be eroded. Non-zero (True) elements form the subset to be eroded.

structure : array_like, optional
Structuring element used for the erosion. Non-zero elements are considered True. If no structuring element is provided, an element is generated with a square connectivity equal to one.

iterations : {int, float}, optional
The erosion is repeated *iterations* times (one, by default). If iterations is less than 1, the erosion is repeated until the result does not change anymore.

mask : array_like, optional
If a mask is given, only those elements with a True value at the corresponding mask element are modified at each iteration.

output : ndarray, optional
Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin : int or tuple of ints, optional
Placement of the filter, by default 0.

border_value : int (cast to 0 or 1), optional
Value at the border in the output array.

Returns **binary_erosion** : ndarray of bools
Erosion of the input by the structuring element.

See also:

`grey_erosion`, `binary_dilation`, `binary_closing`, `binary_opening`, `generate_binary_structure`

Notes

Erosion *[R137]* is a mathematical morphology operation *[R138]* that uses a structuring element for shrinking the shapes in an image. The binary erosion of an image by a structuring element is the locus of the points where a superimposition of the structuring element centered on the point is entirely contained in the set of non-zero elements of the image.

References

[R137], *[R138]*

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 2:5] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_erosion(a).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
```

```
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> #Erosion removes objects smaller than the structure
>>> ndimage.binary_erosion(a, structure=np.ones((5,5))).astype(a.dtype)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

`scipy.ndimage.`**`binary_fill_holes`**(*input*, *structure=None*, *output=None*, *origin=0*)

Fill the holes in binary objects.

Parameters

input : array_like
: n-dimensional binary array with holes to be filled

structure : array_like, optional
: Structuring element used in the computation; large-size elements make computations faster but may miss holes separated from the background by thin regions. The default element (with a square connectivity equal to one) yields the intuitive result where all holes in the input have been filled.

output : ndarray, optional
: Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin : int, tuple of ints, optional
: Position of the structuring element.

Returns

out : ndarray
: Transformation of the initial image *input* where holes have been filled.

See also:

`binary_dilation`, *`binary_propagation`*, *`label`*

Notes

The algorithm used in this function consists in invading the complementary of the shapes in *input* from the outer boundary of the image, using binary dilations. Holes are not connected to the boundary and are therefore not invaded. The result is the complementary subset of the invaded region.

References

[R139]

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5, 5), dtype=int)
>>> a[1:4, 1:4] = 1
>>> a[2,2] = 0
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> ndimage.binary_fill_holes(a).astype(int)
```

```
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Too big structuring element
>>> ndimage.binary_fill_holes(a, structure=np.ones((5,5))).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
```

scipy.ndimage.**binary_hit_or_miss**(*input*, *structure1=None*, *structure2=None*, *output=None*, *origin1=0*, *origin2=None*)

Multi-dimensional binary hit-or-miss transform.

The hit-or-miss transform finds the locations of a given pattern inside the input image.

Parameters

input : array_like (cast to booleans)
: Binary image where a pattern is to be detected.

structure1 : array_like (cast to booleans), optional
: Part of the structuring element to be fitted to the foreground (non-zero elements) of *input*. If no value is provided, a structure of square connectivity 1 is chosen.

structure2 : array_like (cast to booleans), optional
: Second part of the structuring element that has to miss completely the foreground. If no value is provided, the complementary of *structure1* is taken.

output : ndarray, optional
: Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin1 : int or tuple of ints, optional
: Placement of the first part of the structuring element *structure1*, by default 0 for a centered structure.

origin2 : int or tuple of ints, optional
: Placement of the second part of the structuring element *structure2*, by default 0 for a centered structure. If a value is provided for *origin1* and not for *origin2*, then *origin2* is set to *origin1*.

Returns

binary_hit_or_miss : ndarray
: Hit-or-miss transform of *input* with the given structuring element (*structure1*, *structure2*).

See also:

ndimage.morphology, *binary_erosion*

References

[R140]

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[1, 1] = 1; a[2:4, 2:4] = 1; a[4:6, 4:6] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0],
```

```
       [0, 0, 1, 1, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> structure1 = np.array([[1, 0, 0], [0, 1, 1], [0, 1, 1]])
>>> structure1
array([[1, 0, 0],
       [0, 1, 1],
       [0, 1, 1]])
>>> # Find the matches of structure1 in the array a
>>> ndimage.binary_hit_or_miss(a, structure1=structure1).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # Change the origin of the filter
>>> # origin1=1 is equivalent to origin1=(1,1) here
>>> ndimage.binary_hit_or_miss(a, structure1=structure1,\
... origin1=1).astype(int)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

scipy.ndimage.**binary_opening**(*input*, *structure=None*, *iterations=1*, *output=None*, *origin=0*)

Multi-dimensional binary opening with the given structuring element.

The *opening* of an input image by a structuring element is the *dilation* of the *erosion* of the image by the structuring element.

Parameters

input : array_like
: Binary array_like to be opened. Non-zero (True) elements form the subset to be opened.

structure : array_like, optional
: Structuring element used for the opening. Non-zero elements are considered True. If no structuring element is provided an element is generated with a square connectivity equal to one (i.e., only nearest neighbors are connected to the center, diagonally-connected elements are not considered neighbors).

iterations : {int, float}, optional
: The erosion step of the opening, then the dilation step are each repeated *iterations* times (one, by default). If *iterations* is less than 1, each operation is repeated until the result does not change anymore.

output : ndarray, optional
: Array of the same shape as input, into which the output is placed. By default, a new array is created.

origin : int or tuple of ints, optional
: Placement of the filter, by default 0.

Returns

binary_opening : ndarray of bools
: Opening of the input by the structuring element.

See also:

grey_opening, *binary_closing*, *binary_erosion*, *binary_dilation*, *generate_binary_structure*

Notes

Opening [R141] is a mathematical morphology operation [R142] that consists in the succession of an erosion and a dilation of the input with the same structuring element. Opening therefore removes objects smaller than the structuring element.

Together with *closing* (*binary_closing*), opening can be used for noise removal.

References

[R141], [R142]

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((5,5), dtype=int)
>>> a[1:4, 1:4] = 1; a[4, 4] = 1
>>> a
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 1]])
>>> # Opening removes small objects
>>> ndimage.binary_opening(a, structure=np.ones((3,3))).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening can also smooth corners
>>> ndimage.binary_opening(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
>>> # Opening is the dilation of the erosion of the input
>>> ndimage.binary_erosion(a).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> ndimage.binary_dilation(ndimage.binary_erosion(a)).astype(int)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]])
```

scipy.ndimage.**binary_propagation**(*input*, *structure=None*, *mask=None*, *output=None*, *border_value=0*, *origin=0*)

Multi-dimensional binary propagation with the given structuring element.

Parameters **input** : array_like
Binary image to be propagated inside *mask*.

structure : array_like, optional
Structuring element used in the successive dilations. The output may depend on the structuring element, especially if *mask* has several connex components. If no structuring element is provided, an element is generated with a squared connectivity equal to one.

mask : array_like, optional
Binary mask defining the region into which *input* is allowed to propagate.

output : ndarray, optional
Array of the same shape as input, into which the output is placed. By default, a new array is created.

border_value : int (cast to 0 or 1), optional
Value at the border in the output array.

origin : int or tuple of ints, optional
Placement of the filter, by default 0.

Returns **binary_propagation** : ndarray
Binary propagation of *input* inside *mask*.

Notes

This function is functionally equivalent to calling binary_dilation with the number of iterations less then one: iterative dilation until the result does not change anymore.

The succession of an erosion and propagation inside the original image can be used instead of an *opening* for deleting small objects while keeping the contours of larger objects untouched.

References

[R143], *[R144]*

Examples

```
>>> from scipy import ndimage
>>> input = np.zeros((8, 8), dtype=int)
>>> input[2, 2] = 1
>>> mask = np.zeros((8, 8), dtype=int)
>>> mask[1:4, 1:4] = mask[4, 4]  = mask[6:8, 6:8] = 1
>>> input
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]])
>>> mask
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0, 0, 1, 1]])
>>> ndimage.binary_propagation(input, mask=mask).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
```

```
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_propagation(input, mask=mask,\
... structure=np.ones((3,3))).astype(int)
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 1, 1, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]])
```

```
>>> # Comparison between opening and erosion+propagation
>>> a = np.zeros((6,6), dtype=int)
>>> a[2:5, 2:5] = 1; a[0, 0] = 1; a[5, 5] = 1
>>> a
array([[1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 1]])
>>> ndimage.binary_opening(a).astype(int)
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0]])
>>> b = ndimage.binary_erosion(a)
>>> b.astype(int)
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
>>> ndimage.binary_propagation(b, mask=a).astype(int)
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0]])
```

`scipy.ndimage.`**`black_tophat`**(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional black tophat filter.

Parameters

input : array_like
: Input.

size : tuple of ints, optional
: Shape of a flat and full structuring element used for the filter. Optional if *footprint* or *structure* is provided.

footprint : array of ints, optional
Positions of non-infinite elements of a flat structuring element used for the black tophat filter.
structure : array of ints, optional
Structuring element used for the filter. *structure* may be a non-flat structuring element.
output : array, optional
An array used for storing the output of the filter may be provided.
mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'
cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0

Returns **black_tophat** : ndarray
Result of the filter of *input* with *structure*.

See also:

`white_tophat`, `grey_opening`, `grey_closing`

`scipy.ndimage.`**`distance_transform_bf`**(*input*, *metric='euclidean'*, *sampling=None*, *return_distances=True*, *return_indices=False*, *distances=None*, *indices=None*)

Distance transform function by a brute force algorithm.

This function calculates the distance transform of the *input*, by replacing each background element (zero values), with its shortest distance to the foreground (any element non-zero).

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result.

Parameters **input** : array_like
Input
metric : str, optional
Three types of distance metric are supported: 'euclidean', 'taxicab' and 'chessboard'.
sampling : {int, sequence of ints}, optional
This parameter is only used in the case of the euclidean *metric* distance transform.
The sampling along each axis can be given by the *sampling* parameter which should be a sequence of length equal to the input rank, or a single number in which the *sampling* is assumed to be equal along all axes.
return_distances : bool, optional
The *return_distances* flag can be used to indicate if the distance transform is returned.
The default is True.
return_indices : bool, optional
The *return_indices* flags can be used to indicate if the feature transform is returned.
The default is False.
distances : float64 ndarray, optional
Optional output array to hold distances (if *return_distances* is True).
indices : int64 ndarray, optional
Optional output array to hold indices (if *return_indices* is True).

Returns **distances** : ndarray
Distance array if *return_distances* is True.
indices : ndarray
Indices array if *return_indices* is True.

Notes

This function employs a slow brute force algorithm, see also the function distance_transform_cdt for more efficient taxicab and chessboard algorithms.

`scipy.ndimage.distance_transform_cdt(input, metric='chessboard', return_distances=True, return_indices=False, distances=None, indices=None)`

Distance transform for chamfer type of transforms.

Parameters

input : array_like
Input

metric : {'chessboard', 'taxicab'}, optional
The *metric* determines the type of chamfering that is done. If the *metric* is equal to 'taxicab' a structure is generated using generate_binary_structure with a squared distance equal to 1. If the *metric* is equal to 'chessboard', a *metric* is generated using generate_binary_structure with a squared distance equal to the dimensionality of the array. These choices correspond to the common interpretations of the 'taxicab' and the 'chessboard' distance metrics in two dimensions.
The default for *metric* is 'chessboard'.

return_distances, return_indices : bool, optional
The *return_distances*, and *return_indices* flags can be used to indicate if the distance transform, the feature transform, or both must be returned.
If the feature transform is returned (`return_indices=True`), the index of the closest background element is returned along the first axis of the result.
The *return_distances* default is True, and the *return_indices* default is False.

distances, indices : ndarrays of int32, optional
The *distances* and *indices* arguments can be used to give optional output arrays that must be the same shape as *input*.

`scipy.ndimage.distance_transform_edt(input, sampling=None, return_distances=True, return_indices=False, distances=None, indices=None)`

Exact euclidean distance transform.

In addition to the distance transform, the feature transform can be calculated. In this case the index of the closest background element is returned along the first axis of the result.

Parameters

input : array_like
Input data to transform. Can be any type but will be converted into binary: 1 wherever input equates to True, 0 elsewhere.

sampling : float or int, or sequence of same, optional
Spacing of elements along each dimension. If a sequence, must be of length equal to the input rank; if a single number, this is used for all axes. If not specified, a grid spacing of unity is implied.

return_distances : bool, optional
Whether to return distance matrix. At least one of return_distances/return_indices must be True. Default is True.

return_indices : bool, optional
Whether to return indices matrix. Default is False.

distances : ndarray, optional
Used for output of distance array, must be of type float64.

indices : ndarray, optional
Used for output of indices, must be of type int32.

Returns

distance_transform_edt : ndarray or list of ndarrays
Either distance matrix, index matrix, or a list of the two, depending on *return_x* flags and *distance* and *indices* input parameters.

Notes

The euclidean distance transform gives values of the euclidean distance:

```
               n
y_i = sqrt(sum (x[i]-b[i])**2)
               i
```

where b[i] is the background point (value 0) with the smallest Euclidean distance to input points x[i], and n is the number of dimensions.

Examples

```
>>> from scipy import ndimage
>>> a = np.array(([0,1,1,1,1],
...               [0,0,1,1,1],
...               [0,1,1,1,1],
...               [0,1,1,1,0],
...               [0,1,1,0,0]))
>>> ndimage.distance_transform_edt(a)
array([[ 0.    ,  1.    ,  1.4142,  2.2361,  3.    ],
       [ 0.    ,  0.    ,  1.    ,  2.    ,  2.    ],
       [ 0.    ,  1.    ,  1.4142,  1.4142,  1.    ],
       [ 0.    ,  1.    ,  1.4142,  1.    ,  0.    ],
       [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
```

With a sampling of 2 units along x, 1 along y:

```
>>> ndimage.distance_transform_edt(a, sampling=[2,1])
array([[ 0.    ,  1.    ,  2.    ,  2.8284,  3.6056],
       [ 0.    ,  0.    ,  1.    ,  2.    ,  3.    ],
       [ 0.    ,  1.    ,  2.    ,  2.2361,  2.    ],
       [ 0.    ,  1.    ,  2.    ,  1.    ,  0.    ],
       [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
```

Asking for indices as well:

```
>>> edt, inds = ndimage.distance_transform_edt(a, return_indices=True)
>>> inds
array([[[0, 0, 1, 1, 3],
        [1, 1, 1, 1, 3],
        [2, 2, 1, 3, 3],
        [3, 3, 4, 4, 3],
        [4, 4, 4, 4, 4]],
       [[0, 0, 1, 1, 4],
        [0, 1, 1, 1, 4],
        [0, 0, 1, 4, 4],
        [0, 0, 3, 3, 4],
        [0, 0, 3, 3, 4]]])
```

With arrays provided for inplace outputs:

```
>>> indices = np.zeros(((np.ndim(a),) + a.shape), dtype=np.int32)
>>> ndimage.distance_transform_edt(a, return_indices=True, indices=indices)
array([[ 0.    ,  1.    ,  1.4142,  2.2361,  3.    ],
       [ 0.    ,  0.    ,  1.    ,  2.    ,  2.    ],
       [ 0.    ,  1.    ,  1.4142,  1.4142,  1.    ],
       [ 0.    ,  1.    ,  1.4142,  1.    ,  0.    ],
       [ 0.    ,  1.    ,  1.    ,  0.    ,  0.    ]])
```

```
>>> indices
array([[[0, 0, 1, 1, 3],
        [1, 1, 1, 1, 3],
        [2, 2, 1, 3, 3],
        [3, 3, 4, 4, 3],
        [4, 4, 4, 4, 4]],
       [[0, 0, 1, 1, 4],
        [0, 1, 1, 1, 4],
        [0, 0, 1, 4, 4],
        [0, 0, 3, 3, 4],
        [0, 0, 3, 3, 4]]])
```

`scipy.ndimage.`**`generate_binary_structure`**(*rank*, *connectivity*)

Generate a binary structure for binary morphological operations.

Parameters

rank : int
: Number of dimensions of the array to which the structuring element will be applied, as returned by *np.ndim*.

connectivity : int
: *connectivity* determines which elements of the output array belong to the structure, i.e. are considered as neighbors of the central element. Elements up to a squared distance of *connectivity* from the center are considered neighbors. *connectivity* may range from 1 (no diagonal elements are neighbors) to *rank* (all elements are neighbors).

Returns

output : ndarray of bools
: Structuring element which may be used for binary morphological operations, with *rank* dimensions and all dimensions equal to 3.

See also:

iterate_structure, *binary_dilation*, *binary_erosion*

Notes

generate_binary_structure can only create structuring elements with dimensions equal to 3, i.e. minimal dimensions. For larger structuring elements, that are useful e.g. for eroding large objects, one may either use *iterate_structure*, or create directly custom arrays with numpy functions such as `numpy.ones`.

Examples

```
>>> from scipy import ndimage
>>> struct = ndimage.generate_binary_structure(2, 1)
>>> struct
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> a = np.zeros((5,5))
>>> a[2, 2] = 1
>>> a
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> b = ndimage.binary_dilation(a, structure=struct).astype(a.dtype)
>>> b
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
```

```
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> ndimage.binary_dilation(b, structure=struct).astype(a.dtype)
array([[ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 0.,  1.,  1.,  1.,  0.],
       [ 0.,  0.,  1.,  0.,  0.]])
>>> struct = ndimage.generate_binary_structure(2, 2)
>>> struct
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
>>> struct = ndimage.generate_binary_structure(3, 1)
>>> struct # no diagonal elements
array([[[False, False, False],
        [False,  True, False],
        [False, False, False]],
       [[False,  True, False],
        [ True,  True,  True],
        [False,  True, False]],
       [[False, False, False],
        [False,  True, False],
        [False, False, False]]], dtype=bool)
```

`scipy.ndimage.`**`grey_closing`**(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional greyscale closing.

A greyscale closing consists in the succession of a greyscale dilation, and a greyscale erosion.

Parameters

input : array_like
: Array over which the grayscale closing is to be computed.

size : tuple of ints
: Shape of a flat and full structuring element used for the grayscale closing. Optional if *footprint* or *structure* is provided.

footprint : array of ints, optional
: Positions of non-infinite elements of a flat structuring element used for the grayscale closing.

structure : array of ints, optional
: Structuring element used for the grayscale closing. *structure* may be a non-flat structuring element.

output : array, optional
: An array used for storing the ouput of the closing may be provided.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
: The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
: Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

origin : scalar, optional
: The *origin* parameter controls the placement of the filter. Default 0

Returns

grey_closing : ndarray
: Result of the grayscale closing of *input* with *structure*.

See also:

binary_closing, *grey_dilation*, *grey_erosion*, *grey_opening*, *generate_binary_structure*

Notes

The action of a grayscale closing with a flat structuring element amounts to smoothen deep local minima, whereas binary closing fills small holes.

References

[R145]

Examples

```
>>> from scipy import ndimage
>>> a = np.arange(36).reshape((6,6))
>>> a[3,3] = 0
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20,  0, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
>>> ndimage.grey_closing(a, size=(3,3))
array([[ 7,  7,  8,  9, 10, 11],
       [ 7,  7,  8,  9, 10, 11],
       [13, 13, 14, 15, 16, 17],
       [19, 19, 20, 20, 22, 23],
       [25, 25, 26, 27, 28, 29],
       [31, 31, 32, 33, 34, 35]])
>>> # Note that the local minimum a[3,3] has disappeared
```

scipy.ndimage.**grey_dilation**(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Calculate a greyscale dilation, using either a structuring element, or a footprint corresponding to a flat structuring element.

Grayscale dilation is a mathematical morphology operation. For the simple case of a full and flat structuring element, it can be viewed as a maximum filter over a sliding window.

Parameters

input : array_like
Array over which the grayscale dilation is to be computed.

size : tuple of ints
Shape of a flat and full structuring element used for the grayscale dilation. Optional if *footprint* or *structure* is provided.

footprint : array of ints, optional
Positions of non-infinite elements of a flat structuring element used for the grayscale dilation. Non-zero values give the set of neighbors of the center over which the maximum is chosen.

structure : array of ints, optional
Structuring element used for the grayscale dilation. *structure* may be a non-flat structuring element.

output : array, optional
An array used for storing the ouput of the dilation may be provided.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0

Returns **grey_dilation** : ndarray
Grayscale dilation of *input*.

See also:

binary_dilation, *grey_erosion*, *grey_closing*, *grey_opening*, *generate_binary_structure*, ndimage.maximum_filter

Notes

The grayscale dilation of an image input by a structuring element s defined over a domain E is given by:

(input+s)(x) = max {input(y) + s(x-y), for y in E}

In particular, for structuring elements defined as s(y) = 0 for y in E, the grayscale dilation computes the maximum of the input image inside a sliding window defined by E.

Grayscale dilation *[R146]* is a *mathematical morphology* operation *[R147]*.

References

[R146], *[R147]*

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[2:5, 2:5] = 1
>>> a[4,4] = 2; a[2,3] = 3
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 3, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_dilation(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 3, 3, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_dilation(a, footprint=np.ones((3,3)))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 3, 3, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> s = ndimage.generate_binary_structure(2,1)
>>> s
array([[False,  True, False],
```

```
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> ndimage.grey_dilation(a, footprint=s)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 3, 1, 0, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 1, 3, 2, 1, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 0, 1, 1, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_dilation(a, size=(3,3), structure=np.ones((3,3)))
array([[1, 1, 1, 1, 1, 1, 1],
       [1, 2, 4, 4, 4, 2, 1],
       [1, 2, 4, 4, 4, 2, 1],
       [1, 2, 4, 4, 4, 3, 1],
       [1, 2, 2, 3, 3, 3, 1],
       [1, 2, 2, 3, 3, 3, 1],
       [1, 1, 1, 1, 1, 1, 1]])
```

`scipy.ndimage.`**`grey_erosion`**`(`*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*`)`

Calculate a greyscale erosion, using either a structuring element, or a footprint corresponding to a flat structuring element.

Grayscale erosion is a mathematical morphology operation. For the simple case of a full and flat structuring element, it can be viewed as a minimum filter over a sliding window.

Parameters **input** : array_like

Array over which the grayscale erosion is to be computed.

size : tuple of ints

Shape of a flat and full structuring element used for the grayscale erosion. Optional if *footprint* or *structure* is provided.

footprint : array of ints, optional

Positions of non-infinite elements of a flat structuring element used for the grayscale erosion. Non-zero values give the set of neighbors of the center over which the minimum is chosen.

structure : array of ints, optional

Structuring element used for the grayscale erosion. *structure* may be a non-flat structuring element.

output : array, optional

An array used for storing the ouput of the erosion may be provided.

mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional

The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional

Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

origin : scalar, optional

The *origin* parameter controls the placement of the filter. Default 0

Returns **output** : ndarray

Grayscale erosion of *input*.

See also:

binary_erosion, *grey_dilation*, *grey_opening*, *grey_closing*, *generate_binary_structure*, `ndimage.minimum_filter`

Notes

The grayscale erosion of an image input by a structuring element s defined over a domain E is given by:

(input+s)(x) = min {input(y) - s(x-y), for y in E}

In particular, for structuring elements defined as s(y) = 0 for y in E, the grayscale erosion computes the minimum of the input image inside a sliding window defined by E.

Grayscale erosion *[R148]* is a *mathematical morphology* operation *[R149]*.

References

[R148], *[R149]*

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[1:6, 1:6] = 3
>>> a[4,4] = 2; a[2,3] = 1
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 1, 3, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 3, 3, 3, 2, 3, 0],
       [0, 3, 3, 3, 3, 3, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.grey_erosion(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> footprint = ndimage.generate_binary_structure(2, 1)
>>> footprint
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True, False]], dtype=bool)
>>> # Diagonally-connected elements are not considered neighbors
>>> ndimage.grey_erosion(a, size=(3,3), footprint=footprint)
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 3, 1, 2, 0, 0],
       [0, 0, 3, 2, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

`scipy.ndimage.`**`grey_opening`**(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional greyscale opening.

A greyscale opening consists in the succession of a greyscale erosion, and a greyscale dilation.

Parameters

input : array_like
: Array over which the grayscale opening is to be computed.

size : tuple of ints

Shape of a flat and full structuring element used for the grayscale opening. Optional if *footprint* or *structure* is provided.

footprint : array of ints, optional
Positions of non-infinite elements of a flat structuring element used for the grayscale opening.

structure : array of ints, optional
Structuring element used for the grayscale opening. *structure* may be a non-flat structuring element.

output : array, optional
An array used for storing the ouput of the opening may be provided.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0

Returns **grey_opening** : ndarray
Result of the grayscale opening of *input* with *structure*.

See also:

`binary_opening`, `grey_dilation`, `grey_erosion`, `grey_closing`, `generate_binary_structure`

Notes

The action of a grayscale opening with a flat structuring element amounts to smoothen high local maxima, whereas binary opening erases small objects.

References

[R150]

Examples

```
>>> from scipy import ndimage
>>> a = np.arange(36).reshape((6,6))
>>> a[3, 3] = 50
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 50, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
>>> ndimage.grey_opening(a, size=(3,3))
array([[ 0,  1,  2,  3,  4,  4],
       [ 6,  7,  8,  9, 10, 10],
       [12, 13, 14, 15, 16, 16],
       [18, 19, 20, 22, 22, 22],
       [24, 25, 26, 27, 28, 28],
       [24, 25, 26, 27, 28, 28]])
>>> # Note that the local maximum a[3,3] has disappeared
```

`scipy.ndimage.`**`iterate_structure`**(*structure*, *iterations*, *origin=None*)
Iterate a structure by dilating it with itself.

Parameters **structure** : array_like
Structuring element (an array of bools, for example), to be dilated with itself.
iterations : int
number of dilations performed on the structure with itself
origin : optional
If origin is None, only the iterated structure is returned. If not, a tuple of the iterated structure and the modified origin is returned.

Returns **iterate_structure** : ndarray of bools
A new structuring element obtained by dilating *structure* (*iterations* - 1) times with itself.

See also:

`generate_binary_structure`

Examples

```
>>> from scipy import ndimage
>>> struct = ndimage.generate_binary_structure(2, 1)
>>> struct.astype(int)
array([[0, 1, 0],
       [1, 1, 1],
       [0, 1, 0]])
>>> ndimage.iterate_structure(struct, 2).astype(int)
array([[0, 0, 1, 0, 0],
       [0, 1, 1, 1, 0],
       [1, 1, 1, 1, 1],
       [0, 1, 1, 1, 0],
       [0, 0, 1, 0, 0]])
>>> ndimage.iterate_structure(struct, 3).astype(int)
array([[0, 0, 0, 1, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 1, 0, 0, 0]])
```

`scipy.ndimage.`**`morphological_gradient`**(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional morphological gradient.

The morphological gradient is calculated as the difference between a dilation and an erosion of the input with a given structuring element.

Parameters **input** : array_like
Array over which to compute the morphlogical gradient.
size : tuple of ints
Shape of a flat and full structuring element used for the mathematical morphology operations. Optional if *footprint* or *structure* is provided. A larger *size* yields a more blurred gradient.
footprint : array of ints, optional
Positions of non-infinite elements of a flat structuring element used for the morphology operations. Larger footprints give a more blurred morphological gradient.
structure : array of ints, optional
Structuring element used for the morphology operations. *structure* may be a non-flat structuring element.
output : array, optional

An array used for storing the ouput of the morphological gradient may be provided.

mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'

cval : scalar, optional
Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

origin : scalar, optional
The *origin* parameter controls the placement of the filter. Default 0

Returns **morphological_gradient** : ndarray
Morphological gradient of *input*.

See also:

`grey_dilation`, `grey_erosion`, `ndimage.gaussian_gradient_magnitude`

Notes

For a flat structuring element, the morphological gradient computed at a given point corresponds to the maximal difference between elements of the input among the elements covered by the structuring element centered on the point.

References

[R155]

Examples

```
>>> from scipy import ndimage
>>> a = np.zeros((7,7), dtype=int)
>>> a[2:5, 2:5] = 1
>>> ndimage.morphological_gradient(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> # The morphological gradient is computed as the difference
>>> # between a dilation and an erosion
>>> ndimage.grey_dilation(a, size=(3,3)) -\
...  ndimage.grey_erosion(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 0]])
>>> a = np.zeros((7,7), dtype=int)
>>> a[2:5, 2:5] = 1
>>> a[4,4] = 2; a[2,3] = 3
>>> a
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 3, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
```

```
       [0, 0, 0, 0, 0, 0, 0]])
>>> ndimage.morphological_gradient(a, size=(3,3))
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 3, 3, 1, 0],
       [0, 1, 3, 2, 3, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 1, 1, 2, 2, 2, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

`scipy.ndimage.morphological_laplace`(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional morphological laplace.

Parameters **input** : array_like
  Input.
size : int or sequence of ints, optional
  See *structure*.
footprint : bool or ndarray, optional
  See *structure*.
structure : structure, optional
  Either *size*, *footprint*, or the *structure* must be provided.
output : ndarray, optional
  An output array can optionally be provided.
mode : {'reflect','constant','nearest','mirror', 'wrap'}, optional
  The mode parameter determines how the array borders are handled. For 'constant' mode, values beyond borders are set to be *cval*. Default is 'reflect'.
cval : scalar, optional
  Value to fill past edges of input if mode is 'constant'. Default is 0.0
origin : origin, optional
  The origin parameter controls the placement of the filter.

Returns **morphological_laplace** : ndarray
  Output

`scipy.ndimage.white_tophat`(*input*, *size=None*, *footprint=None*, *structure=None*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multi-dimensional white tophat filter.

Parameters **input** : array_like
  Input.
size : tuple of ints
  Shape of a flat and full structuring element used for the filter. Optional if *footprint* or *structure* is provided.
footprint : array of ints, optional
  Positions of elements of a flat structuring element used for the white tophat filter.
structure : array of ints, optional
  Structuring element used for the filter. *structure* may be a non-flat structuring element.
output : array, optional
  An array used for storing the output of the filter may be provided.
mode : {'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional
  The *mode* parameter determines how the array borders are handled, where *cval* is the value when mode is equal to 'constant'. Default is 'reflect'
cval : scalar, optional
  Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.
origin : scalar, optional
  The *origin* parameter controls the placement of the filter. Default is 0.

Returns **output** : ndarray
Result of the filter of *input* with *structure*.

See also:

black_tophat

5.16.6 Utility

imread(fname[, flatten, mode])	Read an image from a file as an array.

scipy.ndimage.**imread**(*fname*, *flatten=False*, *mode=None*)
Read an image from a file as an array.

Parameters **fname** : str or file object
The file name or file object to be read.
flatten : bool, optional
If True, flattens the color layers into a single gray-scale layer.
mode : str, optional
Mode to convert image to, e.g. 'RGB'. See the Notes for more details.

Returns **imread** : ndarray
The array obtained by reading the image.

Notes

imread uses the Python Imaging Library (PIL) to read an image. The following notes are from the PIL documentation.

mode can be one of the following strings:

- 'L' (8-bit pixels, black and white)
- 'P' (8-bit pixels, mapped to any other mode using a color palette)
- 'RGB' (3x8-bit pixels, true color)
- 'RGBA' (4x8-bit pixels, true color with transparency mask)
- 'CMYK' (4x8-bit pixels, color separation)
- 'YCbCr' (3x8-bit pixels, color video format)
- 'I' (32-bit signed integer pixels)
- 'F' (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including 'LA' ('L' with alpha), 'RGBX' (true color with padding) and 'RGBa' (true color with premultiplied alpha).

When translating a color image to black and white (mode 'L', 'I' or 'F'), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

When *flatten* is True, the image is converted using mode 'F'. When *mode* is not None and *flatten* is True, the image is first converted according to *mode*, and the result is then flattened using mode 'F'.

5.17 Orthogonal distance regression (scipy.odr)

5.17.1 Package Content

Data(x[, y, we, wd, fix, meta])	The data to fit.
RealData(x[, y, sx, sy, covx, covy, fix, meta])	The data, with weightings as actual standard deviations and/or covariances.
Model(fcn[, fjacb, fjacd, extra_args, ...])	The Model class stores information about the function you wish to fit.
ODR(data, model[, beta0, delta0, ifixb, ...])	The ODR class gathers all information and coordinates the running of the main fitting routine.
Output(output)	The Output class stores the output of an ODR run.
odr(fcn, beta0, y, x[, we, wd, fjacb, ...])	Low-level function for ODR.
OdrWarning	Warning indicating that the data passed into ODR will cause problems when passed into 'odr' that the user should be aware of.
OdrError	Exception indicating an error in fitting.
OdrStop	Exception stopping fitting.
odr_error	alias of *OdrError*
odr_stop	alias of *OdrStop*

class scipy.odr.**Data**(*x*, *y=None*, *we=None*, *wd=None*, *fix=None*, *meta={}*)

The data to fit.

Parameters

x : array_like

Observed data for the independent variable of the regression

y : array_like, optional

If array-like, observed data for the dependent variable of the regression. A scalar input implies that the model to be used on the data is implicit.

we : array_like, optional

If *we* is a scalar, then that value is used for all data points (and all dimensions of the response variable). If *we* is a rank-1 array of length q (the dimensionality of the response variable), then this vector is the diagonal of the covariant weighting matrix for all data points. If *we* is a rank-1 array of length n (the number of data points), then the i'th element is the weight for the i'th response variable observation (single-dimensional only). If *we* is a rank-2 array of shape (q, q), then this is the full covariant weighting matrix broadcast to each observation. If *we* is a rank-2 array of shape (q, n), then *we[:,i]* is the diagonal of the covariant weighting matrix for the i'th observation. If *we* is a rank-3 array of shape (q, q, n), then *we[:,:,i]* is the full specification of the covariant weighting matrix for each observation. If the fit is implicit, then only a positive scalar value is used.

wd : array_like, optional

If *wd* is a scalar, then that value is used for all data points (and all dimensions of the input variable). If *wd* = 0, then the covariant weighting matrix for each observation is set to the identity matrix (so each dimension of each observation has the same weight). If *wd* is a rank-1 array of length m (the dimensionality of the input variable), then this vector is the diagonal of the covariant weighting matrix for all data points. If *wd* is a rank-1 array of length n (the number of data points), then the i'th element is the weight for the i'th input variable observation (single-dimensional only). If *wd* is a rank-2 array of shape (m, m), then this is the full covariant weighting matrix broadcast to each observation. If *wd* is a rank-2 array of shape (m, n), then *wd[:,i]* is the diagonal of the covariant weighting matrix for the i'th observation. If *wd* is a rank-3 array of shape (m,

m, n), then *wd[:,:,i]* is the full specification of the covariant weighting matrix for each observation.

fix : array_like of ints, optional
The *fix* argument is the same as ifixx in the class ODR. It is an array of integers with the same shape as data.x that determines which input observations are treated as fixed. One can use a sequence of length m (the dimensionality of the input observations) to fix some dimensions for all observations. A value of 0 fixes the observation, a value > 0 makes it free.

meta : dict, optional
Free-form dictionary for metadata.

Notes

Each argument is attached to the member of the instance of the same name. The structures of *x* and *y* are described in the Model class docstring. If *y* is an integer, then the Data instance can only be used to fit with implicit models where the dimensionality of the response is equal to the specified value of *y*.

The *we* argument weights the effect a deviation in the response variable has on the fit. The *wd* argument weights the effect a deviation in the input variable has on the fit. To handle multidimensional inputs and responses easily, the structure of these arguments has the n'th dimensional axis first. These arguments heavily use the structured arguments feature of ODRPACK to conveniently and flexibly support all options. See the ODRPACK User's Guide for a full explanation of how these weights are used in the algorithm. Basically, a higher value of the weight for a particular data point makes a deviation at that point more detrimental to the fit.

Methods

`set_meta`(**kwds)	Update the metadata dictionary with the keywords and data provided by keywords.

`Data.set_meta`(***kwds*)
Update the metadata dictionary with the keywords and data provided by keywords.

Examples

```
data.set_meta(lab="Ph 7; Lab 26", title="Ag110 + Ag108 Decay")
```

class `scipy.odr.RealData`(*x*, *y=None*, *sx=None*, *sy=None*, *covx=None*, *covy=None*, *fix=None*, *meta={}*)
The data, with weightings as actual standard deviations and/or covariances.

Parameters

x : array_like
Observed data for the independent variable of the regression

y : array_like, optional
If array-like, observed data for the dependent variable of the regression. A scalar input implies that the model to be used on the data is implicit.

sx : array_like, optional
Standard deviations of *x*. *sx* are standard deviations of *x* and are converted to weights by dividing 1.0 by their squares.

sy : array_like, optional
Standard deviations of *y*. *sy* are standard deviations of *y* and are converted to weights by dividing 1.0 by their squares.

covx : array_like, optional
Covariance of *x* *covx* is an array of covariance matrices of *x* and are converted to weights by performing a matrix inversion on each observation's covariance matrix.

covy : array_like, optional

Covariance of *y covy* is an array of covariance matrices and are converted to weights by performing a matrix inversion on each observation's covariance matrix.

fix : array_like, optional
The argument and member fix is the same as Data.fix and ODR.ifixx: It is an array of integers with the same shape as *x* that determines which input observations are treated as fixed. One can use a sequence of length m (the dimensionality of the input observations) to fix some dimensions for all observations. A value of 0 fixes the observation, a value > 0 makes it free.

meta : dict, optional
Free-form dictionary for metadata.

Notes

The weights *wd* and *we* are computed from provided values as follows:

sx and *sy* are converted to weights by dividing 1.0 by their squares. For example, `wd = 1./numpy.power(`sx`, 2)`.

covx and *covy* are arrays of covariance matrices and are converted to weights by performing a matrix inversion on each observation's covariance matrix. For example, `we[i] = numpy.linalg.inv(covy[i])`.

These arguments follow the same structured argument conventions as wd and we only restricted by their natures: *sx* and *sy* can't be rank-3, but *covx* and *covy* can be.

Only set *either sx* or *covx* (not both). Setting both will raise an exception. Same with *sy* and *covy*.

Methods

set_meta(**kwds)	Update the metadata dictionary with the keywords and data provided by keywords.

RealData.**set_meta**(***kwds*)
Update the metadata dictionary with the keywords and data provided by keywords.

Examples

```
data.set_meta(lab="Ph 7; Lab 26", title="Ag110 + Ag108 Decay")
```

class scipy.odr.**Model**(*fcn*, *fjacb=None*, *fjacd=None*, *extra_args=None*, *estimate=None*, *implicit=0*, *meta=None*)
The Model class stores information about the function you wish to fit.

It stores the function itself, at the least, and optionally stores functions which compute the Jacobians used during fitting. Also, one can provide a function that will provide reasonable starting values for the fit parameters possibly given the set of data.

Parameters

fcn : function
fcn(beta, x) –> y

fjacb : function
Jacobian of fcn wrt the fit parameters beta.
fjacb(beta, x) –> @f_i(x,B)/@B_j

fjacd : function
Jacobian of fcn wrt the (possibly multidimensional) input variable.
fjacd(beta, x) –> @f_i(x,B)/@x_j

extra_args : tuple, optional
If specified, *extra_args* should be a tuple of extra arguments to pass to *fcn*, *fjacb*, and *fjacd*. Each will be called by *apply(fcn, (beta, x) + extra_args)*

estimate : array_like of rank-1
Provides estimates of the fit parameters from the data
estimate(data) –> estbeta

implicit : boolean
If TRUE, specifies that the model is implicit; i.e *fcn(beta, x)* ~= 0 and there is no y data to fit against

meta : dict, optional
freeform dictionary of metadata for the model

Notes

Note that the *fcn*, *fjacb*, and *fjacd* operate on NumPy arrays and return a NumPy array. The *estimate* object takes an instance of the Data class.

Here are the rules for the shapes of the argument and return arrays of the callback functions:

x if the input data is single-dimensional, then *x* is rank-1 array; i.e. `x = array([1, 2, 3, ...]); x.shape = (n,)` If the input data is multi-dimensional, then *x* is a rank-2 array; i.e., `x = array([[1, 2, ...], [2, 4, ...]]); x.shape = (m, n)`. In all cases, it has the same shape as the input data array passed to `odr`. *m* is the dimensionality of the input data, *n* is the number of observations.

y if the response variable is single-dimensional, then *y* is a rank-1 array, i.e., `y = array([2, 4, ...]); y.shape = (n,)`. If the response variable is multi-dimensional, then *y* is a rank-2 array, i.e., `y = array([[2, 4, ...], [3, 6, ...]]); y.shape = (q, n)` where *q* is the dimensionality of the response variable.

beta rank-1 array of length *p* where *p* is the number of parameters; i.e. `beta = array([B_1, B_2, ..., B_p])`

fjacb if the response variable is multi-dimensional, then the return array's shape is *(q, p, n)* such that `fjacb(x,beta)[l,k,i] = d f_l(X,B)/d B_k` evaluated at the i'th data point. If *q == 1*, then the return array is only rank-2 and with shape *(p, n)*.

fjacd as with fjacb, only the return array's shape is *(q, m, n)* such that `fjacd(x,beta)[l,j,i] = d f_l(X,B)/d X_j` at the i'th data point. If *q == 1*, then the return array's shape is *(m, n)*. If *m == 1*, the shape is (q, n). If *m == q == 1*, the shape is *(n,)*.

Methods

`set_meta`(**kwds)	Update the metadata dictionary with the keywords and data provided here.

`Model.`**`set_meta`**`(`***kwds*`)`
Update the metadata dictionary with the keywords and data provided here.

Examples

set_meta(name="Exponential", equation="y = a exp(b x) + c")

class `scipy.odr.`**`ODR`**`(`*data*, *model*, *beta0=None*, *delta0=None*, *ifixb=None*, *ifixx=None*, *job=None*, *iprint=None*, *errfile=None*, *rptfile=None*, *ndigit=None*, *taufac=None*, *sstol=None*, *partol=None*, *maxit=None*, *stpb=None*, *stpd=None*, *sclb=None*, *scld=None*, *work=None*, *iwork=None*`)`

The ODR class gathers all information and coordinates the running of the main fitting routine.

Members of instances of the ODR class have the same names as the arguments to the initialization routine.

Parameters **data** : Data class instance

instance of the Data class

model : Model class instance
instance of the Model class

Other Parameters

beta0 : array_like of rank-1
a rank-1 sequence of initial parameter values. Optional if model provides an "estimate" function to estimate these values.

delta0 : array_like of floats of rank-1, optional
a (double-precision) float array to hold the initial values of the errors in the input variables. Must be same shape as data.x

ifixb : array_like of ints of rank-1, optional
sequence of integers with the same length as beta0 that determines which parameters are held fixed. A value of 0 fixes the parameter, a value > 0 makes the parameter free.

ifixx : array_like of ints with same shape as data.x, optional
an array of integers with the same shape as data.x that determines which input observations are treated as fixed. One can use a sequence of length m (the dimensionality of the input observations) to fix some dimensions for all observations. A value of 0 fixes the observation, a value > 0 makes it free.

job : int, optional
an integer telling ODRPACK what tasks to perform. See p. 31 of the ODRPACK User's Guide if you absolutely must set the value here. Use the method set_job post-initialization for a more readable interface.

iprint : int, optional
an integer telling ODRPACK what to print. See pp. 33-34 of the ODRPACK User's Guide if you absolutely must set the value here. Use the method set_iprint post-initialization for a more readable interface.

errfile : str, optional
string with the filename to print ODRPACK errors to. *Do Not Open This File Yourself!*

rptfile : str, optional
string with the filename to print ODRPACK summaries to. *Do Not Open This File Yourself!*

ndigit : int, optional
integer specifying the number of reliable digits in the computation of the function.

taufac : float, optional
float specifying the initial trust region. The default value is 1. The initial trust region is equal to taufac times the length of the first computed Gauss-Newton step. taufac must be less than 1.

sstol : float, optional
float specifying the tolerance for convergence based on the relative change in the sum-of-squares. The default value is eps**(1/2) where eps is the smallest value such that 1 + eps > 1 for double precision computation on the machine. sstol must be less than 1.

partol : float, optional
float specifying the tolerance for convergence based on the relative change in the estimated parameters. The default value is eps**(2/3) for explicit models and `eps**(1/3)` for implicit models. partol must be less than 1.

maxit : int, optional
integer specifying the maximum number of iterations to perform. For first runs, maxit is the total number of iterations performed and defaults to 50. For restarts, maxit is the number of additional iterations to perform and defaults to 10.

stpb : array_like, optional
sequence (`len(stpb) == len(beta0)`) of relative step sizes to compute finite difference derivatives wrt the parameters.

stpd : optional

array (`stpd.shape == data.x.shape` or `stpd.shape == (m,)`) of relative step sizes to compute finite difference derivatives wrt the input variable errors. If stpd is a rank-1 array with length m (the dimensionality of the input variable), then the values are broadcast to all observations.

sclb : array_like, optional
sequence (`len(stpb) == len(beta0)`) of scaling factors for the parameters. The purpose of these scaling factors are to scale all of the parameters to around unity. Normally appropriate scaling factors are computed if this argument is not specified. Specify them yourself if the automatic procedure goes awry.

scld : array_like, optional
array (scld.shape == data.x.shape or scld.shape == (m,)) of scaling factors for the *errors* in the input variables. Again, these factors are automatically computed if you do not provide them. If scld.shape == (m,), then the scaling factors are broadcast to all observations.

work : ndarray, optional
array to hold the double-valued working data for ODRPACK. When restarting, takes the value of self.output.work.

iwork : ndarray, optional
array to hold the integer-valued working data for ODRPACK. When restarting, takes the value of self.output.iwork.

Attributes

data	(Data) The data for this fit
model	(Model) The model used in fit
output	(Output) An instance if the Output class containing all of the returned data from an invocation of ODR.run() or ODR.restart()

Methods

restart([iter])	Restarts the run with iter more iterations.
run()	Run the fitting routine with all of the information given.
set_iprint([init, so_init, iter, so_iter, ...])	Set the iprint parameter for the printing of computation reports.
set_job([fit_type, deriv, var_calc, ...])	Sets the "job" parameter is a hopefully comprehensible way.

ODR.**restart**(*iter=None*)
Restarts the run with iter more iterations.

Parameters **iter** : int, optional
ODRPACK's default for the number of new iterations is 10.

Returns **output** : Output instance
This object is also assigned to the attribute .output .

ODR.**run**()
Run the fitting routine with all of the information given.

Returns **output** : Output instance
This object is also assigned to the attribute .output .

ODR.**set_iprint**(*init=None*, *so_init=None*, *iter=None*, *so_iter=None*, *iter_step=None*, *final=None*, *so_final=None*)
Set the iprint parameter for the printing of computation reports.

If any of the arguments are specified here, then they are set in the iprint member. If iprint is not set manually or with this method, then ODRPACK defaults to no printing. If no filename is specified with the member rptfile, then ODRPACK prints to stdout. One can tell ODRPACK to print to stdout in addition to the specified filename by setting the so_* arguments to this function, but one cannot specify to print to stdout but not a file since one can do that by not specifying a rptfile filename.

There are three reports: initialization, iteration, and final reports. They are represented by the arguments init, iter, and final respectively. The permissible values are 0, 1, and 2 representing "no report", "short report", and "long report" respectively.

The argument iter_step (0 <= iter_step <= 9) specifies how often to make the iteration report; the report will be made for every iter_step'th iteration starting with iteration one. If iter_step == 0, then no iteration report is made, regardless of the other arguments.

If the rptfile is None, then any so_* arguments supplied will raise an exception.

ODR.**set_job**(*fit_type=None*, *deriv=None*, *var_calc=None*, *del_init=None*, *restart=None*)

Sets the "job" parameter is a hopefully comprehensible way.

If an argument is not specified, then the value is left as is. The default value from class initialization is for all of these options set to 0.

Parameters

fit_type : {0, 1, 2} int
- 0 -> explicit ODR
- 1 -> implicit ODR
- 2 -> ordinary least-squares

deriv : {0, 1, 2, 3} int
- 0 -> forward finite differences
- 1 -> central finite differences
- ***2 -> user-supplied derivatives (Jacobians) with results***
  checked by ODRPACK
- 3 -> user-supplied derivatives, no checking

var_calc : {0, 1, 2} int
- ***0 -> calculate asymptotic covariance matrix and fit***
  parameter uncertainties (V_B, s_B) using derivatives recomputed at the final solution
- 1 -> calculate V_B and s_B using derivatives from last iteration
- 2 -> do not calculate V_B and s_B

del_init : {0, 1} int
- 0 -> initial input variable offsets set to 0
- 1 -> initial offsets provided by user in variable "work"

restart : {0, 1} int
- 0 -> fit is not a restart
- 1 -> fit is a restart

Notes

The permissible values are different from those given on pg. 31 of the ODRPACK User's Guide only in that one cannot specify numbers greater than the last value for each variable.

If one does not supply functions to compute the Jacobians, the fitting procedure will change deriv to 0, finite differences, as a default. To initialize the input variable offsets by yourself, set del_init to 1 and put the offsets into the "work" variable correctly.

class scipy.odr.**Output**(*output*)

The Output class stores the output of an ODR run.

Notes

Takes one argument for initialization, the return value from the function *odr*. The attributes listed as "optional" above are only present if *odr* was run with `full_output=1`.

Attributes

beta	(ndarray) Estimated parameter values, of shape (q,).
sd_beta	(ndarray) Standard errors of the estimated parameters, of shape (p,).
cov_beta	(ndarray) Covariance matrix of the estimated parameters, of shape (p,p).
delta	(ndarray, optional) Array of estimated errors in input variables, of same shape as *x*.
eps	(ndarray, optional) Array of estimated errors in response variables, of same shape as *y*.
xplus	(ndarray, optional) Array of `x + delta`.
y	(ndarray, optional) Array `y = fcn(x + delta)`.
res_var	(float, optional) Residual variance.
sum_sqare	(float, optional) Sum of squares error.
sum_square_delta	(float, optional) Sum of squares of delta error.
sum_square_eps	(float, optional) Sum of squares of eps error.
inv_condnum	(float, optional) Inverse condition number (cf. ODRPACK UG p. 77).
rel_error	(float, optional) Relative error in function values computed within fcn.
work	(ndarray, optional) Final work array.
work_ind	(dict, optional) Indices into work for drawing out values (cf. ODRPACK UG p. 83).
info	(int, optional) Reason for returning, as output by ODRPACK (cf. ODRPACK UG p. 38).
stopreason	(list of str, optional) *info* interpreted into English.

Methods

pprint()	Pretty-print important results.

`Output.`**`pprint`**`()`

Pretty-print important results.

`scipy.odr.`**`odr`**`(`*fcn*, *beta0*, *y*, *x*, *we=None*, *wd=None*, *fjacb=None*, *fjacd=None*, *extra_args=None*, *ifixx=None*, *ifixb=None*, *job=0*, *iprint=0*, *errfile=None*, *rptfile=None*, *ndigit=0*, *taufac=0.0*, *sstol=-1.0*, *partol=-1.0*, *maxit=-1*, *stpb=None*, *stpd=None*, *sclb=None*, *scld=None*, *work=None*, *iwork=None*, *full_output=0*`)`

Low-level function for ODR.

See also:

ODR, *Model*, *Data*, *RealData*

Notes

This is a function performing the same operation as the *ODR*, *Model* and *Data* classes together. The parameters of this function are explained in the class documentation.

exception `scipy.odr.`**`OdrWarning`**

Warning indicating that the data passed into ODR will cause problems when passed into 'odr' that the user should be aware of.

exception `scipy.odr.`**`OdrError`**

Exception indicating an error in fitting.

This is raised by *scipy.odr* if an error occurs during fitting.

exception `scipy.odr.`**`OdrStop`**
Exception stopping fitting.

You can raise this exception in your objective function to tell *`scipy.odr`* to stop fitting.

`scipy.odr.`**`odr_error`**
alias of *`OdrError`*

`scipy.odr.`**`odr_stop`**
alias of *`OdrStop`*

Prebuilt models:

`polynomial`(order)	Factory function for a general polynomial model.

`scipy.odr.`**`polynomial`**(*order*)
Factory function for a general polynomial model.

Parameters **order** : int or sequence
If an integer, it becomes the order of the polynomial to fit. If a sequence of numbers, then these are the explicit powers in the polynomial. A constant term (power 0) is always included, so don't include 0. Thus, polynomial(n) is equivalent to polynomial(range(1, n+1)).

Returns **polynomial** : Model instance
Model instance.

`scipy.odr.`**`exponential`**

`scipy.odr.`**`multilinear`**

`scipy.odr.`**`unilinear`**

`scipy.odr.`**`quadratic`**

`scipy.odr.`**`polynomial`**

5.17.2 Usage information

Introduction

Why Orthogonal Distance Regression (ODR)? Sometimes one has measurement errors in the explanatory (a.k.a., "independent") variable(s), not just the response (a.k.a., "dependent") variable(s). Ordinary Least Squares (OLS) fitting procedures treat the data for explanatory variables as fixed, i.e., not subject to error of any kind. Furthermore, OLS procedures require that the response variables be an explicit function of the explanatory variables; sometimes making the equation explicit is impractical and/or introduces errors. ODR can handle both of these cases with ease, and can even reduce to the OLS case if that is sufficient for the problem.

ODRPACK is a FORTRAN-77 library for performing ODR with possibly non-linear fitting functions. It uses a modified trust-region Levenberg-Marquardt-type algorithm *[R771]* to estimate the function parameters. The fitting functions are provided by Python functions operating on NumPy arrays. The required derivatives may be provided by Python functions as well, or may be estimated numerically. ODRPACK can do explicit or implicit ODR fits, or it can do OLS. Input and output variables may be multi-dimensional. Weights can be provided to account for different variances of the observations, and even covariances between dimensions of the variables.

The `scipy.odr` package offers an object-oriented interface to ODRPACK, in addition to the low-level `odr` function.

Additional background information about ODRPACK can be found in the ODRPACK User's Guide, reading which is recommended.

Basic usage

1. Define the function you want to fit against.:

```
def f(B, x):
    '''Linear function y = m*x + b'''
    # B is a vector of the parameters.
    # x is an array of the current x values.
    # x is in the same format as the x passed to Data or RealData.
    #
    # Return an array in the same format as y passed to Data or RealData.
    return B[0]*x + B[1]
```

2. Create a Model.:

```
linear = Model(f)
```

3. Create a Data or RealData instance.:

```
mydata = Data(x, y, wd=1./power(sx,2), we=1./power(sy,2))
```

or, when the actual covariances are known:

```
mydata = RealData(x, y, sx=sx, sy=sy)
```

4. Instantiate ODR with your data, model and initial parameter estimate.:

```
myodr = ODR(mydata, linear, beta0=[1., 2.])
```

5. Run the fit.:

```
myoutput = myodr.run()
```

6. Examine output.:

```
myoutput.pprint()
```

References

5.18 Optimization and root finding (`scipy.optimize`)

5.18.1 Optimization

Local Optimization

minimize(fun, x0[, args, method, jac, hess, ...])	Minimization of scalar function of one or more variables.
minimize_scalar(fun[, bracket, bounds, ...])	Minimization of scalar function of one variable.

Continued on next page

Table 5.102 – continued from previous page

OptimizeResult	Represents the optimization result.
OptimizeWarning	

scipy.optimize.**minimize**(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

Minimization of scalar function of one or more variables.

In general, the optimization problems are of the form:

```
minimize f(x) subject to

g_i(x) >= 0,  i = 1,...,m
h_j(x)  = 0,  j = 1,...,p
```

where x is a vector of one or more variables. `g_i(x)` are the inequality constraints. `h_j(x)` are the equality constrains.

Optionally, the lower and upper bounds for each element in x can also be specified using the *bounds* argument.

Parameters

fun : callable
Objective function.

x0 : ndarray
Initial guess.

args : tuple, optional
Extra arguments passed to the objective function and its derivatives (Jacobian, Hessian).

method : str or callable, optional
Type of solver. Should be one of

- 'Nelder-Mead' *(see here)*
- 'Powell' *(see here)*
- 'CG' *(see here)*
- 'BFGS' *(see here)*
- 'Newton-CG' *(see here)*
- 'L-BFGS-B' *(see here)*
- 'TNC' *(see here)*
- 'COBYLA' *(see here)*
- 'SLSQP' *(see here)*
- 'dogleg' *(see here)*
- 'trust-ncg' *(see here)*
- custom - a callable object (added in version 0.14.0), see below for description.

If not given, chosen to be one of `BFGS`, `L-BFGS-B`, `SLSQP`, depending if the problem has constraints or bounds.

jac : bool or callable, optional
Jacobian (gradient) of objective function. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg. If *jac* is a Boolean and is True, *fun* is assumed to return the gradient along with the objective function. If False, the gradient will be estimated numerically. *jac* can also be a callable returning the gradient of the objective. In this case, it must accept the same arguments as *fun*.

hess, hessp : callable, optional
Hessian (matrix of second-order derivatives) of objective function or Hessian of objective function times an arbitrary vector p. Only for Newton-CG, dogleg, trust-ncg. Only one of *hessp* or *hess* needs to be given. If *hess* is provided, then *hessp* will be ignored. If neither *hess* nor *hessp* is provided, then the Hessian product will be approximated using finite differences on *jac*. *hessp* must compute the Hessian times an arbitrary vector.

bounds : sequence, optional

Bounds for variables (only for L-BFGS-B, TNC and SLSQP). `(min, max)` pairs for each element in `x`, defining the bounds on that parameter. Use None for one of `min` or `max` when there is no bound in that direction.

constraints : dict or sequence of dict, optional

Constraints definition (only for COBYLA and SLSQP). Each constraint is defined in a dictionary with fields:

- ***type*** [str] Constraint type: 'eq' for equality, 'ineq' for inequality.
- ***fun*** [callable] The function defining the constraint.
- ***jac*** [callable, optional] The Jacobian of *fun* (only for SLSQP).
- ***args*** [sequence, optional] Extra arguments to be passed to the function and Jacobian.

Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative. Note that COBYLA only supports inequality constraints.

tol : float, optional

Tolerance for termination. For detailed control, use solver-specific options.

options : dict, optional

A dictionary of solver options. All methods accept the following generic options:

- ***maxiter*** [int] Maximum number of iterations to perform.
- ***disp*** [bool] Set to True to print convergence messages.

For method-specific options, see *`show_options`*.

callback : callable, optional

Called after each iteration, as `callback(xk)`, where `xk` is the current parameter vector.

Returns **res** : OptimizeResult

The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See *`OptimizeResult`* for a description of other attributes.

See also:

`minimize_scalar`
Interface to minimization algorithms for scalar univariate functions

`show_options`
Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *BFGS*.

Unconstrained minimization

Method *Nelder-Mead* uses the Simplex algorithm *[R174]*, *[R175]*. This algorithm is robust in many applications. However, if numerical computation of derivative can be trusted, other algorithms using the first and/or second derivatives information might be preferred for their better performance in general.

Method *Powell* is a modification of Powell's method *[R176]*, *[R177]* which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken.

Method *CG* uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in *[R178]* pp. 120-122. Only the first derivatives are used.

Method *BFGS* uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) *[R178]* pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess_inv* in the OptimizeResult object.

Method *Newton-CG* uses a Newton-CG algorithm *[R178]* pp. 168 (also known as the truncated Newton method). It uses a CG method to the compute the search direction. See also *TNC* method for a box-constrained minimization with a similar algorithm.

Method *dogleg* uses the dog-leg trust-region algorithm *[R178]* for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method *trust-ncg* uses the Newton conjugate gradient trust-region algorithm *[R178]* for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector.

Constrained minimization

Method *L-BFGS-B* uses the L-BFGS-B algorithm *[R179]*, *[R180]* for bound constrained minimization.

Method *TNC* uses a truncated Newton algorithm *[R178]*, *[R181]* to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

Method *COBYLA* uses the Constrained Optimization BY Linear Approximation (COBYLA) method *[R182]*,[10],[11]. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm. The constraints functions 'fun' may return either a single number or an array or list of numbers.

Method *SLSQP* uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft[12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

Custom minimizers

It may be useful to pass a custom minimization method, for example when using a frontend to this method such as `scipy.optimize.basinhopping` or a different library. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, x0, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as *callback*, *hess*, etc.), except the *options* dict, which has its contents also passed as *method* parameters pair by pair. Also, if *jac* has been passed as a bool type, *jac* and *fun* are mangled so that *fun* returns just the function values and *jac* is converted to a function returning the Jacobian. The method shall return an `OptimizeResult` object.

The provided *method* callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

New in version 0.11.0.

References

[R174], *[R175]*, *[R176]*, *[R177]*, *[R178]*, *[R179]*, *[R180]*, *[R181]*, *[R182]*,[10],[11],[12]

[10] Powell M J D. Direct search algorithms for optimization calculations. 1998. Acta Numerica 7: 287-336.

[11] Powell M J D. A view of algorithms for optimization without derivatives. 2007.Cambridge University Technical Report DAMTP 2007/NA03

[12] Kraft, D. A software package for sequential quadratic programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center – Institute for Flight Mechanics, Koln, Germany.

Examples

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in *rosen* (resp. *rosen_der*, *rosen_hess*) in the *scipy.optimize*.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the *Nelder-Mead* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
```

Now using the *BFGS* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 26
         Function evaluations: 31
         Gradient evaluations: 31
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377],  # may␣
↪vary
       [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
       [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
       [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
       [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,   1.53713523]])
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from *[R178]*). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
...                constraints=cons)
```

It should converge to the theoretical solution (1.4 ,1.7).

`scipy.optimize.`**`minimize_scalar`**(*fun*, *bracket=None*, *bounds=None*, *args=()*, *method='brent'*, *tol=None*, *options=None*)

Minimization of scalar function of one variable.

Parameters

fun : callable
: Objective function. Scalar function, must return a scalar.

bracket : sequence, optional
: For methods 'brent' and 'golden', *`bracket`* defines the bracketing interval and can either have three items `(a, b, c)` so that `a < b < c` and `fun(b) < fun(a), fun(c)` or two items `a` and `c` which are assumed to be a starting interval for a downhill bracket search (see *`bracket`*); it doesn't always mean that the obtained solution will satisfy `a <= x <= c`.

bounds : sequence, optional
: For method 'bounded', *bounds* is mandatory and must have two items corresponding to the optimization bounds.

args : tuple, optional
: Extra arguments passed to the objective function.

method : str or callable, optional
: Type of solver. Should be one of:
  - 'Brent' *(see here)*
  - 'Bounded' *(see here)*
  - 'Golden' *(see here)*
  - custom - a callable object (added in version 0.14.0), see below

tol : float, optional
: Tolerance for termination. For detailed control, use solver-specific options.

options : dict, optional
: A dictionary of solver options.

maxiter	[int] Maximum number of iterations to perform.
disp	[bool] Set to True to print convergence messages.

  See *`show_options`* for solver-specific options.

Returns

res : OptimizeResult
: The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See *`OptimizeResult`* for a description of other attributes.

See also:

`minimize` Interface to minimization algorithms for scalar multivariate functions

`show_options`
Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *Brent*.

Method *Brent* uses Brent's algorithm to find a local minimum. The algorithm uses inverse parabolic interpolation when possible to speed up convergence of the golden section method.

Method *Golden* uses the golden section search technique. It uses analog of the bisection method to decrease the bracketed interval. It is usually preferable to use the *Brent* method.

Method *Bounded* can perform bounded minimization. It uses the Brent method to find a local minimum in the interval x1 < xopt < x2.

Custom minimizers

It may be useful to pass a custom minimization method, for example when using some library frontend to minimize_scalar. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to *minimize* (such as *bracket*, *tol*, etc.), except the *options* dict, which has its contents also passed as *method* parameters pair by pair. The method shall return an `OptimizeResult` object.

The provided *method* callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by *minimize* may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

New in version 0.11.0.

Examples

Consider the problem of minimizing the following function.

```
>>> def f(x):
...     return (x - 2) * x * (x + 2)**2
```

Using the *Brent* method, we find the local minimum as:

```
>>> from scipy.optimize import minimize_scalar
>>> res = minimize_scalar(f)
>>> res.x
1.28077640403
```

Using the *Bounded* method, we find a local minimum with specified bounds as:

```
>>> res = minimize_scalar(f, bounds=(-3, -1), method='bounded')
>>> res.x
-2.0000002026
```

class `scipy.optimize.`**`OptimizeResult`**

Represents the optimization result.

Notes

There may be additional attributes not listed above depending of the specific solver. Since this class is essentially a subclass of dict with attribute accessors, one can see which attributes are available using the *keys()* method.

Attributes

x	(ndarray) The solution of the optimization.
success	(bool) Whether or not the optimizer exited successfully.
status	(int) Termination status of the optimizer. Its value depends on the underlying solver. Refer to *message* for details.
message	(str) Description of the cause of the termination.
fun, jac, hess: ndarray	Values of objective function, its Jacobian and its Hessian (if available). The Hessians may be approximations, see the documentation of the function in question.
hess_inv	(object) Inverse of the objective function's Hessian; may be an approximation. Not available for all solvers. The type of this attribute may be either np.ndarray or scipy.sparse.linalg.LinearOperator.
nfev, njev, nhev	(int) Number of evaluations of the objective functions and of its Jacobian and Hessian.
nit	(int) Number of iterations performed by the optimizer.
maxcv	(float) The maximum constraint violation.

Methods

clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys(...)	v defaults to None.
get((k[,d]) -> D[k] if k in D, ...)	
has_key((k) -> True if D has a key k, else False)	
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values(() -> list of D's values)	
viewitems(...)	
viewkeys(...)	
viewvalues(...)	

OptimizeResult.**clear**() → None. Remove all items from D.

OptimizeResult.**copy**() → a shallow copy of D

OptimizeResult.**fromkeys**(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

OptimizeResult.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

OptimizeResult.**has_key**(*k*) → True if D has a key k, else False

OptimizeResult.**items**() → list of D's (key, value) pairs, as 2-tuples

OptimizeResult.**iteritems**() → an iterator over the (key, value) items of D

OptimizeResult.**iterkeys**() → an iterator over the keys of D

OptimizeResult.**itervalues**() → an iterator over the values of D

OptimizeResult.**keys**() → list of D's keys

OptimizeResult.**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

OptimizeResult.**popitem**() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

OptimizeResult.**setdefault**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

OptimizeResult.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

OptimizeResult.**values**() → list of D's values

OptimizeResult.**viewitems**() → a set-like object providing a view on D's items

OptimizeResult.**viewkeys**() → a set-like object providing a view on D's keys

OptimizeResult.**viewvalues**() → an object providing a view on D's values

exception scipy.optimize.**OptimizeWarning**

The *minimize* function supports the following methods:

minimize(method='Nelder-Mead')

scipy.optimize.**minimize**(*fun*, *x0*, *args=()*, *method='Nelder-Mead'*, *tol=None*, *callback=None*, *options={'disp': False*, *'initial_simplex': None*, *'maxiter': None*, *'xatol': 0.0001*, *'return_all': False*, *'fatol': 0.0001*, *'func': None*, *'maxfev': None}*)

Minimization of scalar function of one or more variables using the Nelder-Mead algorithm.

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize*

Options

disp : bool
Set to True to print convergence messages.

maxiter, maxfev : int
Maximum allowed number of iterations and function evaluations. Will default to `N*200`, where `N` is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.

initial_simplex : array_like of shape (N + 1, N)
Initial simplex. If given, overrides *x0*. `initial_simplex[j,:]` should contain the coordinates of the j-th vertex of the `N+1` vertices in the simplex, where `N` is the dimension.

xatol : float, optional
Absolute error in xopt between iterations that is acceptable for convergence.

fatol : number, optional
Absolute error in func(xopt) between iterations that is acceptable for convergence.

minimize(method='Powell')

`scipy.optimize.minimize` (*fun*, *x0*, *args=()*, *method='Powell'*, *tol=None*, *callback=None*, *options={'disp': False*, *'return_all': False*, *'maxiter': None*, *'direc': None*, *'func': None*, *'maxfev': None*, *'xtol': 0.0001*, *'ftol': 0.0001}*)

Minimization of scalar function of one or more variables using the modified Powell algorithm.

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize*

Options

disp : bool
Set to True to print convergence messages.

xtol : float
Relative error in solution *xopt* acceptable for convergence.

ftol : float
Relative error in `fun(xopt)` acceptable for convergence.

maxiter, maxfev : int
Maximum allowed number of iterations and function evaluations. Will default to `N*1000`, where `N` is the number of variables, if neither *maxiter* or *maxfev* is set. If both *maxiter* and *maxfev* are set, minimization will stop at the first reached.

direc : ndarray
Initial set of direction vectors for the Powell method.

minimize(method='CG')

`scipy.optimize.minimize` (*fun*, *x0*, *args=()*, *method='CG'*, *jac=None*, *tol=None*, *callback=None*, *options={'disp': False*, *'gtol': 1e-05*, *'eps': 1.4901161193847656e-08*, *'return_all': False*, *'maxiter': None*, *'norm': inf}*)

Minimization of scalar function of one or more variables using the conjugate gradient algorithm.

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize*

Options

disp : bool
Set to True to print convergence messages.

maxiter : int
Maximum number of iterations to perform.

gtol : float
Gradient norm must be less than *gtol* before successful termination.

norm : float
Order of norm (Inf is max, -Inf is min).

eps : float or ndarray
If *jac* is approximated, use this value for the step size.

minimize(method='BFGS')

`scipy.optimize.`**`minimize`** (*fun*, *x0*, *args=()*, *method='BFGS'*, *jac=None*, *tol=None*, *callback=None*, *options={'disp': False*, *'gtol': 1e-05*, *'eps': 1.4901161193847656e-08*, *'return_all': False*, *'maxiter': None*, *'norm': inf}*)

Minimization of scalar function of one or more variables using the BFGS algorithm.

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize`*

Options

disp : bool
Set to True to print convergence messages.

maxiter : int
Maximum number of iterations to perform.

gtol : float
Gradient norm must be less than *gtol* before successful termination.

norm : float
Order of norm (Inf is max, -Inf is min).

eps : float or ndarray
If *jac* is approximated, use this value for the step size.

minimize(method='Newton-CG')

`scipy.optimize.`**`minimize`** (*fun*, *x0*, *args=()*, *method='Newton-CG'*, *jac=None*, *hess=None*, *hessp=None*, *tol=None*, *callback=None*, *options={'disp': False*, *'xtol': 1e-05*, *'eps': 1.4901161193847656e-08*, *'return_all': False*, *'maxiter': None}*)

Minimization of scalar function of one or more variables using the Newton-CG algorithm.

Note that the *jac* parameter (Jacobian) is required.

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize`*

Options

disp : bool
Set to True to print convergence messages.

xtol : float
Average relative error in solution *xopt* acceptable for convergence.

maxiter : int
Maximum number of iterations to perform.

eps : float or ndarray
If *jac* is approximated, use this value for the step size.

minimize(method='L-BFGS-B')

`scipy.optimize.`**`minimize`** (*fun*, *x0*, *args=()*, *method='L-BFGS-B'*, *jac=None*, *bounds=None*, *tol=None*, *callback=None*, *options={'disp': None*, *'maxls': 20*, *'iprint': -1*, *'gtol': 1e-05*, *'eps': 1e-08*, *'maxiter': 15000*, *'ftol': 2.220446049250313e-09*, *'maxcor': 10*, *'maxfun': 15000}*)

Minimize a scalar function of one or more variables using the L-BFGS-B algorithm.

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize`*

Options

disp : bool
Set to True to print convergence messages.

maxcor : int
The maximum number of variable metric corrections used to define the limited memory matrix. (The limited memory BFGS method does not store the full hessian but uses this many terms in an approximation to it.)

factr : float
The iteration stops when `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr * eps`, where `eps` is the machine precision, which is automatically generated by the code. Typical values for *factr* are: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy.

ftol : float
The iteration stops when `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= ftol`.

gtol : float
The iteration will stop when `max{|proj g_i | i = 1, ..., n} <= gtol` where `pg_i` is the i-th component of the projected gradient.

eps : float
Step size used for numerical approximation of the jacobian.

disp : int
Set to True to print convergence messages.

maxfun : int
Maximum number of function evaluations.

maxiter : int
Maximum number of iterations.

maxls : int, optional
Maximum number of line search steps (per iteration). Default is 20.

minimize(method='TNC')

`scipy.optimize.minimize` (*fun*, *x0*, *args=()*, *method='TNC'*, *jac=None*, *bounds=None*, *tol=None*, *callback=None*, *options={'disp': False*, *'minfev': 0*, *'scale': None*, *'rescale': -1*, *'offset': None*, *'gtol': -1*, *'eps': 1e-08*, *'eta': -1*, *'maxiter': None*, *'maxCGit': -1*, *'mesg_num': None*, *'ftol': -1*, *'xtol': -1*, *'stepmx': 0*, *'accuracy': 0}*)

Minimize a scalar function of one or more variables using a truncated Newton (TNC) algorithm.

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize*

Options

eps : float
Step size used for numerical approximation of the jacobian.

scale : list of floats
Scaling factors to apply to each variable. If None, the factors are up-low for interval bounded variables and 1+|x] fo the others. Defaults to None

offset : float
Value to subtract from each variable. If None, the offsets are (up+low)/2 for interval bounded variables and x for the others.

disp : bool
Set to True to print convergence messages.

maxCGit : int
Maximum number of hessian*vector evaluations per main iteration. If maxCGit == 0, the direction chosen is -gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -1.

maxiter : int
Maximum number of function evaluation. if None, *maxiter* is set to max(100, 10*len(x0)). Defaults to None.

eta : float
Severity of the line search. if < 0 or > 1, set to 0.25. Defaults to -1.

stepmx : float
Maximum step for the line search. May be increased during call. If too small, it will be set to 10.0. Defaults to 0.

accuracy : float
Relative precision for finite difference calculations. If <= machine_precision, set to sqrt(machine_precision). Defaults to 0.

minfev : float
Minimum function value estimate. Defaults to 0.

ftol : float
Precision goal for the value of f in the stoping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -1.

xtol : float
Precision goal for the value of x in the stopping criterion (after applying x scaling factors). If xtol < 0.0, xtol is set to sqrt(machine_precision). Defaults to -1.

gtol : float
Precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors). If gtol < 0.0, gtol is set to 1e-2 * sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.

rescale : float
Scaling factor (in log10) used to trigger f value rescaling. If 0, rescale at each iteration. If a large value, never rescale. If < 0, rescale is set to 1.3.

minimize(method='COBYLA')

`scipy.optimize.`**`minimize`**(*fun, x0, args=(), method='COBYLA', constraints=(), tol=None, callback=None, options={'iprint': 1, 'disp': False, 'maxiter': 1000, 'catol': 0.0002, 'rhobeg': 1.0}*)

Minimize a scalar function of one or more variables using the Constrained Optimization BY Linear Approximation (COBYLA) algorithm.

See also:

For documentation for the rest of the parameters, see *`scipy.optimize.minimize`*

Options

rhobeg : float
Reasonable initial changes to the variables.

tol : float
Final accuracy in the optimization (not precisely guaranteed). This is a lower bound on the size of the trust region.

disp : bool
Set to True to print convergence messages. If False, *verbosity* is ignored as set to 0.

maxiter : int
Maximum number of function evaluations.

catol : float
Tolerance (absolute) for constraint violations

minimize(method='SLSQP')

`scipy.optimize.`**`minimize`**(*fun, x0, args=(), method='SLSQP', jac=None, bounds=None, constraints=(), tol=None, callback=None, options={'disp': False, 'iprint': 1, 'eps': 1.4901161193847656e-08, 'func': None, 'maxiter': 100, 'ftol': 1e-06}*)

Minimize a scalar function of one or more variables using Sequential Least SQuares Programming (SLSQP).

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

ftol : float
  Precision goal for the value of f in the stopping criterion.
eps : float
  Step size used for numerical approximation of the jacobian.
disp : bool
  Set to True to print convergence messages. If False, *verbosity* is ignored and set to 0.
maxiter : int
  Maximum number of iterations.

minimize(method='dogleg')

`scipy.optimize.`**`minimize`** (*fun*, *x0*, *args=()*, *method='dogleg'*, *jac=None*, *hess=None*, *tol=None*, *callback=None*, *options={}*)
Minimization of scalar function of one or more variables using the dog-leg trust-region algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

initial_trust_radius : float
  Initial trust-region radius.
max_trust_radius : float
  Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.
eta : float
  Trust region related acceptance stringency for proposed steps.
gtol : float
  Gradient norm must be less than *gtol* before successful termination.

minimize(method='trust-ncg')

`scipy.optimize.`**`minimize`** (*fun*, *x0*, *args=()*, *method='trust-ncg'*, *jac=None*, *hess=None*, *hessp=None*, *tol=None*, *callback=None*, *options={}*)
Minimization of scalar function of one or more variables using the Newton conjugate gradient trust-region algorithm.

See also:

For documentation for the rest of the parameters, see `scipy.optimize.minimize`

Options

initial_trust_radius : float
  Initial trust-region radius.
max_trust_radius : float
  Maximum value of the trust-region radius. No steps that are longer than this value will be proposed.
eta : float
  Trust region related acceptance stringency for proposed steps.
gtol : float
  Gradient norm must be less than *gtol* before successful termination.

The `minimize_scalar` function supports the following methods:

minimize_scalar(method='brent')

scipy.optimize.**minimize_scalar**(*fun*, *args=()*, *method='brent'*, *tol=None*, *options={'xtol': 1.48e-08, 'brack': None, 'func': None, 'maxiter': 500}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize_scalar*

Options

maxiter : int
Maximum number of iterations to perform.

xtol : float
Relative error in solution *xopt* acceptable for convergence.

Notes

Uses inverse parabolic interpolation when possible to speed up convergence of golden section method.

minimize_scalar(method='bounded')

scipy.optimize.**minimize_scalar**(*fun*, *bounds=None*, *args=()*, *method='bounded'*, *tol=None*, *options={'disp': 0, 'maxiter': 500, 'func': None, 'xatol': 1e-05}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize_scalar*

Options

maxiter : int
Maximum number of iterations to perform.

disp : bool
Set to True to print convergence messages.

xatol : float
Absolute error in solution *xopt* acceptable for convergence.

minimize_scalar(method='golden')

scipy.optimize.**minimize_scalar**(*fun*, *args=()*, *method='golden'*, *tol=None*, *options={'xtol': 1.4901161193847656e-08, 'brack': None, 'func': None, 'maxiter': 5000}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.minimize_scalar*

Options

maxiter : int
Maximum number of iterations to perform.

xtol : float
Relative error in solution *xopt* acceptable for convergence.

The specific optimization method interfaces below in this subsection are not recommended for use in new scripts; all of these methods are accessible via a newer, more consistent interface provided by the functions above.

General-purpose multivariate methods:

fmin(func, x0[, args, xtol, ftol, maxiter, ...])	Minimize a function using the downhill simplex algorithm.
fmin_powell(func, x0[, args, xtol, ftol, ...])	Minimize a function using modified Powell's method.

Continued on next page

Table 5.104 – continued from previous page

fmin_cg(f, x0[, fprime, args, gtol, norm, ...])	Minimize a function using a nonlinear conjugate gradient algorithm.
fmin_bfgs(f, x0[, fprime, args, gtol, norm, ...])	Minimize a function using the BFGS algorithm.
fmin_ncg(f, x0, fprime[, fhess_p, fhess, ...])	Unconstrained minimization of a function using the Newton-CG method.

scipy.optimize.**fmin**(*func*, *x0*, *args=()*, *xtol=0.0001*, *ftol=0.0001*, *maxiter=None*, *maxfun=None*, *full_output=0*, *disp=1*, *retall=0*, *callback=None*, *initial_simplex=None*)

Minimize a function using the downhill simplex algorithm.

This algorithm only uses function values, not derivatives or second derivatives.

Parameters

func : callable func(x,*args)
The objective function to be minimized.

x0 : ndarray
Initial guess.

args : tuple, optional
Extra arguments passed to func, i.e. `f(x,*args)`.

xtol : float, optional
Absolute error in xopt between iterations that is acceptable for convergence.

ftol : number, optional
Absolute error in func(xopt) between iterations that is acceptable for convergence.

maxiter : int, optional
Maximum number of iterations to perform.

maxfun : number, optional
Maximum number of function evaluations to make.

full_output : bool, optional
Set to True if fopt and warnflag outputs are desired.

disp : bool, optional
Set to True to print convergence messages.

retall : bool, optional
Set to True to return list of solutions at each iteration.

callback : callable, optional
Called after each iteration, as callback(xk), where xk is the current parameter vector.

initial_simplex : array_like of shape (N + 1, N), optional
Initial simplex. If given, overrides *x0*. `initial_simplex[j,:]` should contain the coordinates of the j-th vertex of the `N+1` vertices in the simplex, where `N` is the dimension.

Returns

xopt : ndarray
Parameter that minimizes function.

fopt : float
Value of function at minimum: `fopt = func(xopt)`.

iter : int
Number of iterations performed.

funcalls : int
Number of function calls made.

warnflag : int
1 : Maximum number of function evaluations made. 2 : Maximum number of iterations reached.

allvecs : list
Solution at each iteration.

See also:

minimize Interface to minimization algorithms for multivariate functions. See the 'Nelder-Mead' *method* in particular.

Notes

Uses a Nelder-Mead simplex algorithm to find the minimum of function of one or more variables.

This algorithm has a long history of successful use in applications. But it will usually be slower than an algorithm that uses first or second derivative information. In practice it can have poor performance in high-dimensional problems and is not robust to minimizing complicated functions. Additionally, there currently is no complete theory describing when the algorithm will successfully converge to the minimum, or how fast it will if it does. Both the ftol and xtol criteria must be met for convergence.

References

[R168], *[R169]*

scipy.optimize.**fmin_powell**(*func*, *x0*, *args=()*, *xtol=0.0001*, *ftol=0.0001*, *maxiter=None*, *maxfun=None*, *full_output=0*, *disp=1*, *retall=0*, *callback=None*, *direc=None*)

Minimize a function using modified Powell's method. This method only uses function values, not derivatives.

Parameters

func : callable f(x,*args)
    Objective function to be minimized.

x0 : ndarray
    Initial guess.

args : tuple, optional
    Extra arguments passed to func.

callback : callable, optional
    An optional user-supplied function, called after each iteration. Called as `callback(xk)`, where `xk` is the current parameter vector.

direc : ndarray, optional
    Initial direction set.

xtol : float, optional
    Line-search error tolerance.

ftol : float, optional
    Relative error in `func(xopt)` acceptable for convergence.

maxiter : int, optional
    Maximum number of iterations to perform.

maxfun : int, optional
    Maximum number of function evaluations to make.

full_output : bool, optional
    If True, fopt, xi, direc, iter, funcalls, and warnflag are returned.

disp : bool, optional
    If True, print convergence messages.

retall : bool, optional
    If True, return a list of the solution at each iteration.

Returns

xopt : ndarray
    Parameter which minimizes *func*.

fopt : number
    Value of function at minimum: `fopt = func(xopt)`.

direc : ndarray
    Current direction set.

iter : int
    Number of iterations.

funcalls : int
    Number of function calls made.

warnflag : int
Integer warning flag:
1 : Maximum number of function evaluations. 2 : Maximum number of iterations.
allvecs : list
List of solutions at each iteration.

See also:

minimize Interface to unconstrained minimization algorithms for multivariate functions. See the 'Powell' *method* in particular.

Notes

Uses a modification of Powell's method to find the minimum of a function of N variables. Powell's method is a conjugate direction method.

The algorithm has two loops. The outer loop merely iterates over the inner loop. The inner loop minimizes over each current direction in the direction set. At the end of the inner loop, if certain conditions are met, the direction that gave the largest decrease is dropped and replaced with the difference between the current estimated x and the estimated x from the beginning of the inner-loop.

The technical conditions for replacing the direction of greatest increase amount to checking that

1. No further gain can be made along the direction of greatest increase from that iteration.
2. The direction of greatest increase accounted for a large sufficient fraction of the decrease in the function value from that iteration of the inner loop.

References

Powell M.J.D. (1964) An efficient method for finding the minimum of a function of several variables without calculating derivatives, Computer Journal, 7 (2):155-162.

Press W., Teukolsky S.A., Vetterling W.T., and Flannery B.P.: Numerical Recipes (any edition), Cambridge University Press

scipy.optimize.**fmin_cg**(*f*, *x0*, *fprime=None*, *args=()*, *gtol=1e-05*, *norm=inf*, *epsilon=1.4901161193847656e-08*, *maxiter=None*, *full_output=0*, *disp=1*, *retall=0*, *callback=None*)

Minimize a function using a nonlinear conjugate gradient algorithm.

Parameters

f : callable, `f(x, *args)`
Objective function to be minimized. Here *x* must be a 1-D array of the variables that are to be changed in the search for a minimum, and `args` are the other (fixed) parameters of *f*.
x0 : ndarray
A user-supplied initial estimate of *xopt*, the optimal value of *x*. It must be a 1-D array of values.
fprime : callable, `fprime(x, *args)`, optional
A function that returns the gradient of *f* at *x*. Here *x* and `args` are as described above for *f*. The returned value must be a 1-D array. Defaults to None, in which case the gradient is approximated numerically (see *epsilon*, below).
args : tuple, optional
Parameter values passed to *f* and *fprime*. Must be supplied whenever additional fixed parameters are needed to completely specify the functions *f* and *fprime*.
gtol : float, optional
Stop when the norm of the gradient is less than *gtol*.
norm : float, optional

Order to use for the norm of the gradient (`-np.Inf` is min, `np.Inf` is max).

epsilon : float or ndarray, optional
Step size(s) to use when *fprime* is approximated numerically. Can be a scalar or a 1-D array. Defaults to `sqrt(eps)`, with eps the floating point machine precision. Usually `sqrt(eps)` is about 1.5e-8.

maxiter : int, optional
Maximum number of iterations to perform. Default is `200 * len(x0)`.

full_output : bool, optional
If True, return *fopt*, *func_calls*, *grad_calls*, and *warnflag* in addition to *xopt*. See the Returns section below for additional information on optional return values.

disp : bool, optional
If True, return a convergence message, followed by *xopt*.

retall : bool, optional
If True, add to the returned values the results of each iteration.

callback : callable, optional
An optional user-supplied function, called after each iteration. Called as `callback(xk)`, where `xk` is the current value of *x0*.

Returns **xopt** : ndarray
Parameters which minimize f, i.e. `f(xopt) == fopt`.

fopt : float, optional
Minimum value found, f(xopt). Only returned if *full_output* is True.

func_calls : int, optional
The number of function_calls made. Only returned if *full_output* is True.

grad_calls : int, optional
The number of gradient calls made. Only returned if *full_output* is True.

warnflag : int, optional
Integer value with warning status, only returned if *full_output* is True.
0 : Success.
1 : The maximum number of iterations was exceeded.
2 [Gradient and/or function calls were not changing. May indicate] that precision was lost, i.e., the routine did not converge.

allvecs : list of ndarray, optional
List of arrays, containing the results at each iteration. Only returned if *retall* is True.

See also:

`minimize` common interface to all *`scipy.optimize`* algorithms for unconstrained and constrained minimization of multivariate functions. It provides an alternative way to call `fmin_cg`, by specifying `method='CG'`.

Notes

This conjugate gradient algorithm is based on that of Polak and Ribiere *[R170]*.

Conjugate gradient methods tend to work better when:

1.*f* has a unique global minimizing point, and no local minima or other stationary points,

2.*f* is, at least locally, reasonably well approximated by a quadratic function of the variables,

3.*f* is continuous and has a continuous gradient,

4.*fprime* is not too large, e.g., has a norm less than 1000,

5.The initial guess, *x0*, is reasonably close to *f* 's global minimizing point, *xopt*.

References

[R170]

Examples

Example 1: seek the minimum value of the expression `a*u**2 + b*u*v + c*v**2 + d*u + e*v + f` for given values of the parameters and an initial guess `(u, v) = (0, 0)`.

```
>>> args = (2, 3, 7, 8, 9, 10)  # parameter values
>>> def f(x, *args):
...     u, v = x
...     a, b, c, d, e, f = args
...     return a*u**2 + b*u*v + c*v**2 + d*u + e*v + f
>>> def gradf(x, *args):
...     u, v = x
...     a, b, c, d, e, f = args
...     gu = 2*a*u + b*v + d     # u-component of the gradient
...     gv = b*u + 2*c*v + e     # v-component of the gradient
...     return np.asarray((gu, gv))
>>> x0 = np.asarray((0, 0))  # Initial guess.
>>> from scipy import optimize
>>> res1 = optimize.fmin_cg(f, x0, fprime=gradf, args=args)
Optimization terminated successfully.
         Current function value: 1.617021
         Iterations: 4
         Function evaluations: 8
         Gradient evaluations: 8
>>> res1
array([-1.80851064, -0.25531915])
```

Example 2: solve the same problem using the *minimize* function. (This *myopts* dictionary shows all of the available options, although in practice only non-default values would be needed. The returned value will be a dictionary.)

```
>>> opts = {'maxiter' : None,    # default value.
...         'disp' : True,    # non-default value.
...         'gtol' : 1e-5,    # default value.
...         'norm' : np.inf,  # default value.
...         'eps' : 1.4901161193847656e-08}  # default value.
>>> res2 = optimize.minimize(f, x0, jac=gradf, args=args,
...                          method='CG', options=opts)
Optimization terminated successfully.
        Current function value: 1.617021
        Iterations: 4
        Function evaluations: 8
        Gradient evaluations: 8
>>> res2.x  # minimum found
array([-1.80851064, -0.25531915])
```

`scipy.optimize.`**`fmin_bfgs`**(*f, x0, fprime=None, args=(), gtol=1e-05, norm=inf, epsilon=1.4901161193847656e-08, maxiter=None, full_output=0, disp=1, retall=0, callback=None*)

Minimize a function using the BFGS algorithm.

Parameters

f : callable f(x,*args)
Objective function to be minimized.

x0 : ndarray
Initial guess.

fprime : callable f'(x,*args), optional
Gradient of f.
args : tuple, optional
Extra arguments passed to f and fprime.
gtol : float, optional
Gradient norm must be less than gtol before successful termination.
norm : float, optional
Order of norm (Inf is max, -Inf is min)
epsilon : int or ndarray, optional
If fprime is approximated, use this value for the step size.
callback : callable, optional
An optional user-supplied function to call after each iteration. Called as callback(xk), where xk is the current parameter vector.
maxiter : int, optional
Maximum number of iterations to perform.
full_output : bool, optional
If True,return fopt, func_calls, grad_calls, and warnflag in addition to xopt.
disp : bool, optional
Print convergence message if True.
retall : bool, optional
Return a list of results at each iteration if True.

Returns

xopt : ndarray
Parameters which minimize f, i.e. f(xopt) == fopt.
fopt : float
Minimum value.
gopt : ndarray
Value of gradient at minimum, f'(xopt), which should be near 0.
Bopt : ndarray
Value of 1/f''(xopt), i.e. the inverse hessian matrix.
func_calls : int
Number of function_calls made.
grad_calls : int
Number of gradient calls made.
warnflag : integer
1 : Maximum number of iterations exceeded. 2 : Gradient and/or function calls not changing.
allvecs : list
`OptimizeResult` at each iteration. Only returned if retall is True.

See also:

`minimize` Interface to minimization algorithms for multivariate functions. See the 'BFGS' *method* in particular.

Notes

Optimize the function, f, whose gradient is given by fprime using the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS)

References

Wright, and Nocedal 'Numerical Optimization', 1999, pg. 198.

scipy.optimize.**fmin_ncg**(*f*, *x0*, *fprime*, *fhess_p=None*, *fhess=None*, *args=()*, *avextol=1e-05*, *epsilon=1.4901161193847656e-08*, *maxiter=None*, *full_output=0*, *disp=1*, *retall=0*, *callback=None*)

Unconstrained minimization of a function using the Newton-CG method.

Parameters

f : callable `f(x, *args)`
  Objective function to be minimized.

x0 : ndarray
  Initial guess.

fprime : callable `f'(x, *args)`
  Gradient of f.

fhess_p : callable `fhess_p(x, p, *args)`, optional
  Function which computes the Hessian of f times an arbitrary vector, p.

fhess : callable `fhess(x, *args)`, optional
  Function to compute the Hessian matrix of f.

args : tuple, optional
  Extra arguments passed to f, fprime, fhess_p, and fhess (the same set of extra arguments is supplied to all of these functions).

epsilon : float or ndarray, optional
  If fhess is approximated, use this value for the step size.

callback : callable, optional
  An optional user-supplied function which is called after each iteration. Called as callback(xk), where xk is the current parameter vector.

avextol : float, optional
  Convergence is assumed when the average relative error in the minimizer falls below this amount.

maxiter : int, optional
  Maximum number of iterations to perform.

full_output : bool, optional
  If True, return the optional outputs.

disp : bool, optional
  If True, print convergence message.

retall : bool, optional
  If True, return a list of results at each iteration.

Returns

xopt : ndarray
  Parameters which minimize f, i.e. `f(xopt) == fopt`.

fopt : float
  Value of the function at xopt, i.e. `fopt = f(xopt)`.

fcalls : int
  Number of function calls made.

gcalls : int
  Number of gradient calls made.

hcalls : int
  Number of hessian calls made.

warnflag : int
  Warnings generated by the algorithm. 1 : Maximum number of iterations exceeded.

allvecs : list
  The result at each iteration, if retall is True (see below).

See also:

`minimize` Interface to minimization algorithms for multivariate functions. See the 'Newton-CG' *method* in particular.

Notes

Only one of *fhess_p* or *fhess* need to be given. If *fhess* is provided, then *fhess_p* will be ignored. If neither *fhess* nor *fhess_p* is provided, then the hessian product will be approximated using finite differences on *fprime*. *fhess_p* must compute the hessian times an arbitrary vector. If it is not given, finite-differences on *fprime* are used to compute it.

Newton-CG methods are also called truncated Newton methods. This function differs from scipy.optimize.fmin_tnc because

1. ***scipy.optimize.fmin_ncg is written purely in python using numpy***
   and scipy while scipy.optimize.fmin_tnc calls a C function.

2. ***scipy.optimize.fmin_ncg is only for unconstrained minimization***
   while scipy.optimize.fmin_tnc is for unconstrained minimization or box constrained minimization. (Box constraints give lower and upper bounds for each variable separately.)

References

Wright & Nocedal, 'Numerical Optimization', 1999, pg. 140.

Constrained multivariate methods:

fmin_l_bfgs_b(func, x0[, fprime, args, ...])	Minimize a function func using the L-BFGS-B algorithm.
fmin_tnc(func, x0[, fprime, args, ...])	Minimize a function with variables subject to bounds, using gradient information in a truncated Newton algorithm.
fmin_cobyla(func, x0, cons[, args, ...])	Minimize a function using the Constrained Optimization BY Linear Approximation (COBYLA) method.
fmin_slsqp(func, x0[, eqcons, f_eqcons, ...])	Minimize a function using Sequential Least SQuares Programming
differential_evolution(func, bounds[, args, ...])	Finds the global minimum of a multivariate function.

scipy.optimize.**fmin_l_bfgs_b**(*func*, *x0*, *fprime=None*, *args=()*, *approx_grad=0*, *bounds=None*, *m=10*, *factr=10000000.0*, *pgtol=1e-05*, *epsilon=1e-08*, *iprint=-1*, *maxfun=15000*, *maxiter=15000*, *disp=None*, *callback=None*, *maxls=20*)

Minimize a function func using the L-BFGS-B algorithm.

Parameters

func : callable f(x,*args)
: Function to minimise.

x0 : ndarray
: Initial guess.

fprime : callable fprime(x,*args), optional
: The gradient of *func*. If None, then *func* returns the function value and the gradient (`f, g = func(x, *args)`), unless *approx_grad* is True in which case *func* returns only `f`.

args : sequence, optional
: Arguments to pass to *func* and *fprime*.

approx_grad : bool, optional
: Whether to approximate the gradient numerically (in which case *func* returns only the function value).

bounds : list, optional
: `(min, max)` pairs for each element in `x`, defining the bounds on that parameter. Use None or +-inf for one of `min` or `max` when there is no bound in that direction.

m : int, optional
: The maximum number of variable metric corrections used to define the limited memory matrix. (The limited memory BFGS method does not store the full hessian but uses this

many terms in an approximation to it.)

factr : float, optional

The iteration stops when `(f^k - f^{k+1})/max{|f^k|,|f^{k+1}|,1} <= factr * eps`, where `eps` is the machine precision, which is automatically generated by the code. Typical values for *factr* are: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy.

pgtol : float, optional

The iteration will stop when `max{|proj g_i | i = 1, ..., n} <= pgtol` where `pg_i` is the i-th component of the projected gradient.

epsilon : float, optional

Step size used when *approx_grad* is True, for numerically calculating the gradient

iprint : int, optional

Controls the frequency of output. `iprint < 0` means no output; `iprint = 0` print only one line at the last iteration; `0 < iprint < 99` print also f and `|proj g|` every iprint iterations; `iprint = 99` print details of every iteration except n-vectors; `iprint = 100` print also the changes of active set and final x; `iprint > 100` print details of every iteration including x and g.

disp : int, optional

If zero, then no output. If a positive number, then this over-rides *iprint* (i.e., *iprint* gets the value of *disp*).

maxfun : int, optional

Maximum number of function evaluations.

maxiter : int, optional

Maximum number of iterations.

callback : callable, optional

Called after each iteration, as `callback(xk)`, where `xk` is the current parameter vector.

maxls : int, optional

Maximum number of line search steps (per iteration). Default is 20.

Returns

x : array_like

Estimated position of the minimum.

f : float

Value of *func* at the minimum.

d : dict

Information dictionary.

- d['warnflag'] is
  - 0 if converged,
  - 1 if too many function evaluations or too many iterations,
  - 2 if stopped for another reason, given in d['task']
- d['grad'] is the gradient at the minimum (should be 0 ish)
- d['funcalls'] is the number of function calls made.
- d['nit'] is the number of iterations.

See also:

`minimize` Interface to minimization algorithms for multivariate functions. See the 'L-BFGS-B' *method* in particular.

Notes

License of L-BFGS-B (FORTRAN code):

The version included here (in fortran code) is 3.0 (released April 25, 2011). It was written by Ciyou Zhu, Richard Byrd, and Jorge Nocedal <nocedal@ece.nwu.edu>. It carries the following condition for use:

This software is freely available, but we expect that all publications describing work using this software, or all commercial products using it, quote at least one of the references given below. This software is released under the BSD License.

References

- R. H. Byrd, P. Lu and J. Nocedal. A Limited Memory Algorithm for Bound Constrained Optimization, (1995), SIAM Journal on Scientific and Statistical Computing, 16, 5, pp. 1190-1208.
- C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization (1997), ACM Transactions on Mathematical Software, 23, 4, pp. 550 - 560.
- J.L. Morales and J. Nocedal. L-BFGS-B: Remark on Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization (2011), ACM Transactions on Mathematical Software, 38, 1.

`scipy.optimize.`**`fmin_tnc`**(*func*, *x0*, *fprime=None*, *args=()*, *approx_grad=0*, *bounds=None*, *epsilon=1e-08*, *scale=None*, *offset=None*, *messages=15*, *maxCGit=-1*, *maxfun=None*, *eta=-1*, *stepmx=0*, *accuracy=0*, *fmin=0*, *ftol=-1*, *xtol=-1*, *pgtol=-1*, *rescale=-1*, *disp=None*, *callback=None*)

Minimize a function with variables subject to bounds, using gradient information in a truncated Newton algorithm. This method wraps a C implementation of the algorithm.

Parameters

func : callable `func(x, *args)`
Function to minimize. Must do one of:

1. Return f and g, where f is the value of the function and g its gradient (a list of floats).
2. Return the function value but supply gradient function separately as *fprime*.
3. Return the function value and set `approx_grad=True`.

If the function returns None, the minimization is aborted.

x0 : array_like
Initial estimate of minimum.

fprime : callable `fprime(x, *args)`, optional
Gradient of *func*. If None, then either *func* must return the function value and the gradient (`f,g = func(x, *args)`) or *approx_grad* must be True.

args : tuple, optional
Arguments to pass to function.

approx_grad : bool, optional
If true, approximate the gradient numerically.

bounds : list, optional
(min, max) pairs for each element in x0, defining the bounds on that parameter. Use None or +/-inf for one of min or max when there is no bound in that direction.

epsilon : float, optional
Used if approx_grad is True. The stepsize in a finite difference approximation for fprime.

scale : array_like, optional
Scaling factors to apply to each variable. If None, the factors are up-low for interval bounded variables and 1+|x| for the others. Defaults to None.

offset : array_like, optional
Value to subtract from each variable. If None, the offsets are (up+low)/2 for interval bounded variables and x for the others.

messages : int, optional
Bit mask used to select messages display during minimization values defined in the MSGS dict. Defaults to MGS_ALL.

disp : int, optional
Integer interface to messages. 0 = no message, 5 = all messages

maxCGit : int, optional

Maximum number of hessian*vector evaluations per main iteration. If maxCGit == 0, the direction chosen is -gradient if maxCGit < 0, maxCGit is set to max(1,min(50,n/2)). Defaults to -1.

maxfun : int, optional
Maximum number of function evaluation. if None, maxfun is set to max(100, 10*len(x0)). Defaults to None.

eta : float, optional
Severity of the line search. if < 0 or > 1, set to 0.25. Defaults to -1.

stepmx : float, optional
Maximum step for the line search. May be increased during call. If too small, it will be set to 10.0. Defaults to 0.

accuracy : float, optional
Relative precision for finite difference calculations. If <= machine_precision, set to sqrt(machine_precision). Defaults to 0.

fmin : float, optional
Minimum function value estimate. Defaults to 0.

ftol : float, optional
Precision goal for the value of f in the stoping criterion. If ftol < 0.0, ftol is set to 0.0 defaults to -1.

xtol : float, optional
Precision goal for the value of x in the stopping criterion (after applying x scaling factors). If xtol < 0.0, xtol is set to sqrt(machine_precision). Defaults to -1.

pgtol : float, optional
Precision goal for the value of the projected gradient in the stopping criterion (after applying x scaling factors). If pgtol < 0.0, pgtol is set to 1e-2 * sqrt(accuracy). Setting it to 0.0 is not recommended. Defaults to -1.

rescale : float, optional
Scaling factor (in log10) used to trigger f value rescaling. If 0, rescale at each iteration. If a large value, never rescale. If < 0, rescale is set to 1.3.

callback : callable, optional
Called after each iteration, as callback(xk), where xk is the current parameter vector.

Returns

x : ndarray
The solution.

nfeval : int
The number of function evaluations.

rc : int
Return code, see below

See also:

`minimize` Interface to minimization algorithms for multivariate functions. See the 'TNC' *method* in particular.

Notes

The underlying algorithm is truncated Newton, also called Newton Conjugate-Gradient. This method differs from scipy.optimize.fmin_ncg in that

1. It wraps a C implementation of the algorithm
2. It allows each variable to be given an upper and lower bound.

The algorithm incoporates the bound constraints by determining the descent direction as in an unconstrained truncated Newton, but never taking a step-size large enough to leave the space of feasible x's. The algorithm keeps track of a set of currently active constraints, and ignores them when computing the minimum allowable step size. (The x's associated with the active constraint are kept fixed.) If the maximum allowable step size is

zero then a new constraint is added. At the end of each iteration one of the constraints may be deemed no longer active and removed. A constraint is considered no longer active is if it is currently active but the gradient for that variable points inward from the constraint. The specific constraint removed is the one associated with the variable of largest index whose constraint is no longer active.

Return codes are defined as follows:

```
-1 : Infeasible (lower bound > upper bound)
 0 : Local minimum reached (|pg| ~= 0)
 1 : Converged (|f_n-f_(n-1)| ~= 0)
 2 : Converged (|x_n-x_(n-1)| ~= 0)
 3 : Max. number of function evaluations reached
 4 : Linear search failed
 5 : All lower bounds are equal to the upper bounds
 6 : Unable to progress
 7 : User requested end of minimization
```

References

Wright S., Nocedal J. (2006), 'Numerical Optimization'

Nash S.G. (1984), "Newton-Type Minimization Via the Lanczos Method", SIAM Journal of Numerical Analysis 21, pp. 770-778

scipy.optimize.**fmin_cobyla**(*func*, *x0*, *cons*, *args=()*, *consargs=None*, *rhobeg=1.0*, *rhoend=0.0001*, *iprint=1*, *maxfun=1000*, *disp=None*, *catol=0.0002*)

Minimize a function using the Constrained Optimization BY Linear Approximation (COBYLA) method. This method wraps a FORTRAN implementation of the algorithm.

Parameters

func : callable
: Function to minimize. In the form func(x, *args).

x0 : ndarray
: Initial guess.

cons : sequence
: Constraint functions; must all be `>=0` (a single function if only 1 constraint). Each function takes the parameters *x* as its first argument, and it can return either a single number or an array or list of numbers.

args : tuple, optional
: Extra arguments to pass to function.

consargs : tuple, optional
: Extra arguments to pass to constraint functions (default of None means use same extra arguments as those passed to func). Use `()` for no extra arguments.

rhobeg : float, optional
: Reasonable initial changes to the variables.

rhoend : float, optional
: Final accuracy in the optimization (not precisely guaranteed). This is a lower bound on the size of the trust region.

iprint : {0, 1, 2, 3}, optional
: Controls the frequency of output; 0 implies no output. Deprecated.

disp : {0, 1, 2, 3}, optional
: Over-rides the iprint interface. Preferred.

maxfun : int, optional
: Maximum number of function evaluations.

catol : float, optional
: Absolute tolerance for constraint violations.

Returns

x : ndarray
: The argument that minimises *f*.

See also:

minimize Interface to minimization algorithms for multivariate functions. See the 'COBYLA' *method* in particular.

Notes

This algorithm is based on linear approximations to the objective function and each constraint. We briefly describe the algorithm.

Suppose the function is being minimized over k variables. At the jth iteration the algorithm has k+1 points v_1, ..., v_(k+1), an approximate solution x_j, and a radius RHO_j. (i.e. linear plus a constant) approximations to the objective function and constraint functions such that their function values agree with the linear approximation on the k+1 points v_1,.., v_(k+1). This gives a linear program to solve (where the linear approximations of the constraint functions are constrained to be non-negative).

However the linear approximations are likely only good approximations near the current simplex, so the linear program is given the further requirement that the solution, which will become x_(j+1), must be within RHO_j from x_j. RHO_j only decreases, never increases. The initial RHO_j is rhobeg and the final RHO_j is rhoend. In this way COBYLA's iterations behave like a trust region algorithm.

Additionally, the linear program may be inconsistent, or the approximation may give poor improvement. For details about how these issues are resolved, as well as how the points v_i are updated, refer to the source code or the references below.

References

Powell M.J.D. (1994), "A direct search optimization method that models the objective and constraint functions by linear interpolation.", in Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), pp. 51-67

Powell M.J.D. (1998), "Direct search algorithms for optimization calculations", Acta Numerica 7, 287-336

Powell M.J.D. (2007), "A view of algorithms for optimization without derivatives", Cambridge University Technical Report DAMTP 2007/NA03

Examples

Minimize the objective function f(x,y) = x*y subject to the constraints x**2 + y**2 < 1 and y > 0:

```
>>> def objective(x):
...     return x[0]*x[1]
...
>>> def constr1(x):
...     return 1 - (x[0]**2 + x[1]**2)
...
>>> def constr2(x):
...     return x[1]
...
>>> from scipy.optimize import fmin_cobyla
>>> fmin_cobyla(objective, [0.0, 0.1], [constr1, constr2], rhoend=1e-7)
array([-0.70710685,  0.70710671])
```

The exact solution is (-sqrt(2)/2, sqrt(2)/2).

scipy.optimize.**fmin_slsqp**(*func*, *x0*, *eqcons=()*, *f_eqcons=None*, *ieqcons=()*, *f_ieqcons=None*, *bounds=()*, *fprime=None*, *fprime_eqcons=None*, *fprime_ieqcons=None*, *args=()*, *iter=100*, *acc=1e-06*, *iprint=1*, *disp=None*, *full_output=0*, *epsilon=1.4901161193847656e-08*, *callback=None*)

Minimize a function using Sequential Least SQuares Programming

Python interface function for the SLSQP Optimization subroutine originally implemented by Dieter Kraft.

Parameters

func : callable f(x,*args)
Objective function. Must return a scalar.

x0 : 1-D ndarray of float
Initial guess for the independent variable(s).

eqcons : list, optional
A list of functions of length n such that eqcons[j](x,*args) == 0.0 in a successfully optimized problem.

f_eqcons : callable f(x,*args), optional
Returns a 1-D array in which each element must equal 0.0 in a successfully optimized problem. If f_eqcons is specified, eqcons is ignored.

ieqcons : list, optional
A list of functions of length n such that ieqcons[j](x,*args) >= 0.0 in a successfully optimized problem.

f_ieqcons : callable f(x,*args), optional
Returns a 1-D ndarray in which each element must be greater or equal to 0.0 in a successfully optimized problem. If f_ieqcons is specified, ieqcons is ignored.

bounds : list, optional
A list of tuples specifying the lower and upper bound for each independent variable [(xl0, xu0),(xl1, xu1),...] Infinite values will be interpreted as large floating values.

fprime : callable *f(x,*args)*, optional
A function that evaluates the partial derivatives of func.

fprime_eqcons : callable *f(x,*args)*, optional
A function of the form *f(x, *args)* that returns the m by n array of equality constraint normals. If not provided, the normals will be approximated. The array returned by fprime_eqcons should be sized as (len(eqcons), len(x0)).

fprime_ieqcons : callable *f(x,*args)*, optional
A function of the form *f(x, *args)* that returns the m by n array of inequality constraint normals. If not provided, the normals will be approximated. The array returned by fprime_ieqcons should be sized as (len(ieqcons), len(x0)).

args : sequence, optional
Additional arguments passed to func and fprime.

iter : int, optional
The maximum number of iterations.

acc : float, optional
Requested accuracy.

iprint : int, optional
The verbosity of fmin_slsqp :

- iprint <= 0 : Silent operation
- iprint == 1 : Print summary upon completion (default)
- iprint >= 2 : Print status of each iterate and summary

disp : int, optional
Over-rides the iprint interface (preferred).

full_output : bool, optional
If False, return only the minimizer of func (default). Otherwise, output final objective function and summary information.

epsilon : float, optional
The step size for finite-difference derivative estimates.

callback : callable, optional
Called after each iteration, as `callback(x)`, where `x` is the current parameter vector.

Returns

out : ndarray of float
The final minimizer of func.

fx : ndarray of float, if full_output is true

The final value of the objective function.

its : int, if full_output is true
The number of iterations.

imode : int, if full_output is true
The exit mode from the optimizer (see below).

smode : string, if full_output is true
Message describing the exit mode from the optimizer.

See also:

minimize Interface to minimization algorithms for multivariate functions. See the 'SLSQP' *method* in particular.

Notes

Exit modes are defined as follows

```
-1 : Gradient evaluation required (g & a)
 0 : Optimization terminated successfully.
 1 : Function evaluation required (f & c)
 2 : More equality constraints than independent variables
 3 : More than 3*n iterations in LSQ subproblem
 4 : Inequality constraints incompatible
 5 : Singular matrix E in LSQ subproblem
 6 : Singular matrix C in LSQ subproblem
 7 : Rank-deficient equality constraint subproblem HFTI
 8 : Positive directional derivative for linesearch
 9 : Iteration limit exceeded
```

Examples

Examples are given *in the tutorial*.

`scipy.optimize.`**`differential_evolution`**(*func*, *bounds*, *args=()*, *strategy='best1bin'*, *maxiter=1000*, *popsize=15*, *tol=0.01*, *mutation=(0.5, 1)*, *recombination=0.7*, *seed=None*, *callback=None*, *disp=False*, *polish=True*, *init='latinhypercube'*, *atol=0*)

Finds the global minimum of a multivariate function. Differential Evolution is stochastic in nature (does not use gradient methods) to find the minimium, and can search large areas of candidate space, but often requires larger numbers of function evaluations than conventional gradient based techniques.

The algorithm is due to Storn and Price *[R164]*.

Parameters

func : callable
The objective function to be minimized. Must be in the form `f(x, *args)`, where `x` is the argument in the form of a 1-D array and `args` is a tuple of any additional fixed parameters needed to completely specify the function.

bounds : sequence
Bounds for variables. `(min, max)` pairs for each element in `x`, defining the lower and upper bounds for the optimizing argument of *func*. It is required to have `len(bounds) == len(x)`. `len(bounds)` is used to determine the number of parameters in `x`.

args : tuple, optional
Any additional fixed parameters needed to completely specify the objective function.

strategy : str, optional
The differential evolution strategy to use. Should be one of:

- 'best1bin'

- 'best1exp'
- 'rand1exp'
- 'randtobest1exp'
- 'best2exp'
- 'rand2exp'
- 'randtobest1bin'
- 'best2bin'
- 'rand2bin'
- 'rand1bin'

The default is 'best1bin'.

maxiter : int, optional
The maximum number of generations over which the entire population is evolved. The maximum number of function evaluations (with no polishing) is: `(maxiter + 1) * popsize * len(x)`

popsize : int, optional
A multiplier for setting the total population size. The population has `popsize * len(x)` individuals.

tol : float, optional
Relative tolerance for convergence, the solving stops when `np.std(pop) <= atol + tol * np.abs(np.mean(population_energies))`, where and *atol* and *tol* are the absolute and relative tolerance respectively.

mutation : float or tuple(float, float), optional
The mutation constant. In the literature this is also known as differential weight, being denoted by F. If specified as a float it should be in the range [0, 2]. If specified as a tuple `(min, max)` dithering is employed. Dithering randomly changes the mutation constant on a generation by generation basis. The mutation constant for that generation is taken from `U[min, max)`. Dithering can help speed convergence significantly. Increasing the mutation constant increases the search radius, but will slow down convergence.

recombination : float, optional
The recombination constant, should be in the range [0, 1]. In the literature this is also known as the crossover probability, being denoted by CR. Increasing this value allows a larger number of mutants to progress into the next generation, but at the risk of population stability.

seed : int or *np.random.RandomState*, optional
If *seed* is not specified the *np.RandomState* singleton is used. If *seed* is an int, a new *np.random.RandomState* instance is used, seeded with seed. If *seed* is already a *np.random.RandomState instance*, then that *np.random.RandomState* instance is used. Specify *seed* for repeatable minimizations.

disp : bool, optional
Display status messages

callback : callable, *callback(xk, convergence=val)*, optional
A function to follow the progress of the minimization. `xk` is the current value of `x0`. `val` represents the fractional value of the population convergence. When `val` is greater than one the function halts. If callback returns *True*, then the minimization is halted (any polishing is still carried out).

polish : bool, optional
If True (default), then *scipy.optimize.minimize* with the *L-BFGS-B* method is used to polish the best population member at the end, which can improve the minimization slightly.

init : string, optional
Specify how the population initialization is performed. Should be one of:

- 'latinhypercube'
- 'random'

The default is 'latinhypercube'. Latin Hypercube sampling tries to maximize coverage of the available parameter space. 'random' initializes the population randomly - this has the drawback that clustering can occur, preventing the whole of parameter space being covered.

atol : float, optional
Absolute tolerance for convergence, the solving stops when `np.std(pop) <= atol + tol * np.abs(np.mean(population_energies))`, where and *atol* and *tol* are the absolute and relative tolerance respectively.

Returns **res** : OptimizeResult
The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the optimizer exited successfully and `message` which describes the cause of the termination. See `OptimizeResult` for a description of other attributes. If *polish* was employed, and a lower minimum was obtained by the polishing, then OptimizeResult also contains the `jac` attribute.

Notes

Differential evolution is a stochastic population based method that is useful for global optimization problems. At each pass through the population the algorithm mutates each candidate solution by mixing with other candidate solutions to create a trial candidate. There are several strategies *[R165]* for creating trial candidates, which suit some problems more than others. The 'best1bin' strategy is a good starting point for many systems. In this strategy two members of the population are randomly chosen. Their difference is used to mutate the best member (the *best* in *best1bin*), b_0, so far:

$$b' = b_0 + mutation * (population[rand0] - population[rand1])$$

A trial vector is then constructed. Starting with a randomly chosen 'i'th parameter the trial is sequentially filled (in modulo) with parameters from *b'* or the original candidate. The choice of whether to use *b'* or the original candidate is made with a binomial distribution (the 'bin' in 'best1bin') - a random number in [0, 1) is generated. If this number is less than the *recombination* constant then the parameter is loaded from *b'*, otherwise it is loaded from the original candidate. The final parameter is always loaded from *b'*. Once the trial candidate is built its fitness is assessed. If the trial is better than the original candidate then it takes its place. If it is also better than the best overall candidate it also replaces that. To improve your chances of finding a global minimum use higher *popsize* values, with higher *mutation* and (dithering), but lower *recombination* values. This has the effect of widening the search radius, but slowing convergence.

New in version 0.15.0.

References

[R164], *[R165]*, *[R166]*

Examples

Let us consider the problem of minimizing the Rosenbrock function. This function is implemented in `rosen` in `scipy.optimize`.

```
>>> from scipy.optimize import rosen, differential_evolution
>>> bounds = [(0,2), (0, 2), (0, 2), (0, 2), (0, 2)]
>>> result = differential_evolution(rosen, bounds)
>>> result.x, result.fun
(array([1., 1., 1., 1., 1.]), 1.9216496320061384e-19)
```

Next find the minimum of the Ackley function (http://en.wikipedia.org/wiki/Test_functions_for_optimization).

```
>>> from scipy.optimize import differential_evolution
>>> import numpy as np
>>> def ackley(x):
...     arg1 = -0.2 * np.sqrt(0.5 * (x[0] ** 2 + x[1] ** 2))
...     arg2 = 0.5 * (np.cos(2. * np.pi * x[0]) + np.cos(2. * np.pi * x[1]))
...     return -20. * np.exp(arg1) - np.exp(arg2) + 20. + np.e
>>> bounds = [(-5, 5), (-5, 5)]
>>> result = differential_evolution(ackley, bounds)
>>> result.x, result.fun
(array([ 0.,  0.]), 4.4408920985006262e-16)
```

Univariate (scalar) minimization methods:

`fminbound`(func, x1, x2[, args, xtol, ...])	Bounded minimization for scalar functions.
`brent`(func[, args, brack, tol, full_output, ...])	Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.
`golden`(func[, args, brack, tol, ...])	Return the minimum of a function of one variable.

`scipy.optimize.`**`fminbound`**(*func, x1, x2, args=(), xtol=1e-05, maxfun=500, full_output=0, disp=1*)

Bounded minimization for scalar functions.

Parameters

func : callable f(x,*args)
: Objective function to be minimized (must accept and return scalars).

x1, x2 : float or array scalar
: The optimization bounds.

args : tuple, optional
: Extra arguments passed to function.

xtol : float, optional
: The convergence tolerance.

maxfun : int, optional
: Maximum number of function evaluations allowed.

full_output : bool, optional
: If True, return optional outputs.

disp : int, optional
: ***If non-zero, print messages.***

0 : no message printing. 1 : non-convergence notification messages only. 2 : print a message on convergence too. 3 : print iteration results.

Returns

xopt : ndarray
: Parameters (over given interval) which minimize the objective function.

fval : number
: The function value at the minimum point.

ierr : int
: An error flag (0 if converged, 1 if maximum number of function calls reached).

numfunc : int
: The number of function calls made.

See also:

`minimize_scalar`
Interface to minimization algorithms for scalar univariate functions. See the 'Bounded' *method* in particular.

Notes

Finds a local minimizer of the scalar function *func* in the interval x1 < xopt < x2 using Brent's method. (See `brent` for auto-bracketing).

`scipy.optimize.`**`brent`**(*func*, *args=()*, *brack=None*, *tol=1.48e-08*, *full_output=0*, *maxiter=500*)

Given a function of one-variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.

Parameters

func : callable f(x,*args)
: Objective function.

args : tuple, optional
: Additional arguments (if present).

brack : tuple, optional
: Either a triple (xa,xb,xc) where xa<xb<xc and func(xb) < func(xa), func(xc) or a pair (xa,xb) which are used as a starting interval for a downhill bracket search (see `bracket`). Providing the pair (xa,xb) does not always mean the obtained solution will satisfy xa<=x<=xb.

tol : float, optional
: Stop if between iteration change is less than *tol*.

full_output : bool, optional
: If True, return all output args (xmin, fval, iter, funcalls).

maxiter : int, optional
: Maximum number of iterations in solution.

Returns

xmin : ndarray
: Optimum point.

fval : float
: Optimum value.

iter : int
: Number of iterations.

funcalls : int
: Number of objective function evaluations made.

See also:

`minimize_scalar`
: Interface to minimization algorithms for scalar univariate functions. See the 'Brent' *method* in particular.

Notes

Uses inverse parabolic interpolation when possible to speed up convergence of golden section method.

`scipy.optimize.`**`golden`**(*func*, *args=()*, *brack=None*, *tol=1.4901161193847656e-08*, *full_output=0*, *maxiter=5000*)

Return the minimum of a function of one variable.

Given a function of one variable and a possible bracketing interval, return the minimum of the function isolated to a fractional precision of tol.

Parameters

func : callable func(x,*args)
: Objective function to minimize.

args : tuple, optional
: Additional arguments (if present), passed to func.

brack : tuple, optional
: Triple (a,b,c), where (a<b<c) and func(b) < func(a),func(c). If bracket consists of two numbers (a, c), then they are assumed to be a starting interval for a downhill bracket

search (see `bracket`); it doesn't always mean that obtained solution will satisfy a<=x<=c.

tol : float, optional
x tolerance stop criterion

full_output : bool, optional
If True, return optional outputs.

maxiter : int
Maximum number of iterations to perform.

See also:

`minimize_scalar`
Interface to minimization algorithms for scalar univariate functions. See the 'Golden' *method* in particular.

Notes

Uses analog of bisection method to decrease the bracketed interval.

Equation (Local) Minimizers

`leastsq`(func, x0[, args, Dfun, full_output, ...])	Minimize the sum of squares of a set of equations.
`least_squares`(fun, x0[, jac, bounds, ...])	Solve a nonlinear least-squares problem with bounds on the variables.
`nnls`(A, b)	Solve `argmin_x || Ax - b ||_2` for `x>=0`.
`lsq_linear`(A, b[, bounds, method, tol, ...])	Solve a linear least-squares problem with bounds on the variables.

`scipy.optimize.`**`leastsq`**(*func, x0, args=(), Dfun=None, full_output=0, col_deriv=0, ftol=1.49012e-08, xtol=1.49012e-08, gtol=0.0, maxfev=0, epsfcn=None, factor=100, diag=None*)

Minimize the sum of squares of a set of equations.

```
x = arg min(sum(func(y)**2,axis=0))
         y
```

Parameters **func** : callable
should take at least one (possibly length N vector) argument and returns M floating point numbers. It must not return NaNs or fitting might fail.

x0 : ndarray
The starting estimate for the minimization.

args : tuple, optional
Any extra arguments to func are placed in this tuple.

Dfun : callable, optional
A function or method to compute the Jacobian of func with derivatives across the rows. If this is None, the Jacobian will be estimated.

full_output : bool, optional
non-zero to return all optional outputs.

col_deriv : bool, optional
non-zero to specify that the Jacobian function computes derivatives down the columns (faster, because there is no transpose operation).

ftol : float, optional
Relative error desired in the sum of squares.

xtol : float, optional
Relative error desired in the approximate solution.

gtol : float, optional
Orthogonality desired between the function vector and the columns of the Jacobian.

maxfev : int, optional
The maximum number of calls to the function. If *Dfun* is provided then the default *maxfev* is 100*(N+1) where N is the number of elements in x0, otherwise the default *maxfev* is 200*(N+1).

epsfcn : float, optional
A variable used in determining a suitable step length for the forward- difference approximation of the Jacobian (for Dfun=None). Normally the actual step length will be sqrt(epsfcn)*x If epsfcn is less than the machine precision, it is assumed that the relative errors are of the order of the machine precision.

factor : float, optional
A parameter determining the initial step bound (`factor * || diag * x||`). Should be in interval `(0.1, 100)`.

diag : sequence, optional
N positive entries that serve as a scale factors for the variables.

Returns

x : ndarray
The solution (or the result of the last iteration for an unsuccessful call).

cov_x : ndarray
Uses the fjac and ipvt optional outputs to construct an estimate of the jacobian around the solution. None if a singular matrix encountered (indicates very flat curvature in some direction). This matrix must be multiplied by the residual variance to get the covariance of the parameter estimates – see curve_fit.

infodict : dict
a dictionary of optional outputs with the key s:

nfev	The number of function calls
fvec	The function evaluated at the output
fjac	A permutation of the R matrix of a QR factorization of the final approximate Jacobian matrix, stored column wise. Together with ipvt, the covariance of the estimate can be approximated.
ipvt	An integer array of length N which defines a permutation matrix, p, such that fjac*p = q*r, where r is upper triangular with diagonal elements of nonincreasing magnitude. Column j of p is column ipvt(j) of the identity matrix.
qtf	The vector (transpose(q) * fvec).

mesg : str
A string message giving information about the cause of failure.

ier : int
An integer flag. If it is equal to 1, 2, 3 or 4, the solution was found. Otherwise, the solution was not found. In either case, the optional output variable 'mesg' gives more information.

Notes

"leastsq" is a wrapper around MINPACK's lmdif and lmder algorithms.

cov_x is a Jacobian approximation to the Hessian of the least squares objective function. This approximation assumes that the objective function is based on the difference between some observed target data (ydata) and a (non-linear) function of the parameters *f(xdata, params)*

```
func(params) = ydata - f(xdata, params)
```

so that the objective function is

```
  min   sum((ydata - f(xdata, params))**2, axis=0)
params
```

`scipy.optimize.least_squares`(*fun*, *x0*, *jac='2-point'*, *bounds=(-inf, inf)*, *method='trf'*, *ftol=1e-08*, *xtol=1e-08*, *gtol=1e-08*, *x_scale=1.0*, *loss='linear'*, *f_scale=1.0*, *diff_step=None*, *tr_solver=None*, *tr_options={}*, *jac_sparsity=None*, *max_nfev=None*, *verbose=0*, *args=()*, *kwargs={}*)

Solve a nonlinear least-squares problem with bounds on the variables.

Given the residuals f(x) (an m-dimensional real function of n real variables) and the loss function rho(s) (a scalar function), `least_squares` finds a local minimum of the cost function F(x):

```
minimize F(x) = 0.5 * sum(rho(f_i(x)**2), i = 0, ..., m - 1)
subject to lb <= x <= ub
```

The purpose of the loss function rho(s) is to reduce the influence of outliers on the solution.

Parameters

fun : callable

Function which computes the vector of residuals, with the signature `fun(x, *args, **kwargs)`, i.e., the minimization proceeds with respect to its first argument. The argument `x` passed to this function is an ndarray of shape (n,) (never a scalar, even for n=1). It must return a 1-d array_like of shape (m,) or a scalar. If the argument `x` is complex or the function `fun` returns complex residuals, it must be wrapped in a real function of real arguments, as shown at the end of the Examples section.

x0 : array_like with shape (n,) or float

Initial guess on independent variables. If float, it will be treated as a 1-d array with one element.

jac : {'2-point', '3-point', 'cs', callable}, optional

Method of computing the Jacobian matrix (an m-by-n matrix, where element (i, j) is the partial derivative of f[i] with respect to x[j]). The keywords select a finite difference scheme for numerical estimation. The scheme '3-point' is more accurate, but requires twice as much operations compared to '2-point' (default). The scheme 'cs' uses complex steps, and while potentially the most accurate, it is applicable only when *fun* correctly handles complex inputs and can be analytically continued to the complex plane. Method 'lm' always uses the '2-point' scheme. If callable, it is used as `jac(x, *args, **kwargs)` and should return a good approximation (or the exact value) for the Jacobian as an array_like (np.atleast_2d is applied), a sparse matrix or a `scipy.sparse.linalg.LinearOperator`.

bounds : 2-tuple of array_like, optional

Lower and upper bounds on independent variables. Defaults to no bounds. Each array must match the size of *x0* or be a scalar, in the latter case a bound will be the same for all variables. Use `np.inf` with an appropriate sign to disable bounds on all or some variables.

method : {'trf', 'dogbox', 'lm'}, optional

Algorithm to perform minimization.

- 'trf' : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
- 'dogbox' : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.
- 'lm' : Levenberg-Marquardt algorithm as implemented in MINPACK. Doesn't handle bounds and sparse Jacobians. Usually the most efficient method for small unconstrained problems.

Default is 'trf'. See Notes for more information.

ftol : float, optional
Tolerance for termination by the change of the cost function. Default is 1e-8. The optimization process is stopped when `dF < ftol * F`, and there was an adequate agreement between a local quadratic model and the true model in the last step.

xtol : float, optional
Tolerance for termination by the change of the independent variables. Default is 1e-8. The exact condition depends on the *method* used:

- For 'trf' and 'dogbox' : `norm(dx) < xtol * (xtol + norm(x))`
- For 'lm' : `Delta < xtol * norm(xs)`, where `Delta` is a trust-region radius and `xs` is the value of `x` scaled according to *x_scale* parameter (see below).

gtol : float, optional
Tolerance for termination by the norm of the gradient. Default is 1e-8. The exact condition depends on a *method* used:

- For 'trf' : `norm(g_scaled, ord=np.inf) < gtol`, where `g_scaled` is the value of the gradient scaled to account for the presence of the bounds *[STIR]*.
- For 'dogbox' : `norm(g_free, ord=np.inf) < gtol`, where `g_free` is the gradient with respect to the variables which are not in the optimal state on the boundary.
- For 'lm' : the maximum absolute value of the cosine of angles between columns of the Jacobian and the residual vector is less than *gtol*, or the residual vector is zero.

x_scale : array_like or 'jac', optional
Characteristic scale of each variable. Setting *x_scale* is equivalent to reformulating the problem in scaled variables `xs = x / x_scale`. An alternative view is that the size of a trust region along j-th dimension is proportional to `x_scale[j]`. Improved convergence may be achieved by setting *x_scale* such that a step of a given size along any of the scaled variables has a similar effect on the cost function. If set to 'jac', the scale is iteratively updated using the inverse norms of the columns of the Jacobian matrix (as described in *[JJMore]*).

loss : str or callable, optional
Determines the loss function. The following keyword values are allowed:

- 'linear' (default) : `rho(z) = z`. Gives a standard least-squares problem.
- 'soft_l1' : `rho(z) = 2 * ((1 + z)**0.5 - 1)`. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- 'huber' : `rho(z) = z if z <= 1 else 2*z**0.5 - 1`. Works similarly to 'soft_l1'.
- 'cauchy' : `rho(z) = ln(1 + z)`. Severely weakens outliers influence, but may cause difficulties in optimization process.
- 'arctan' : `rho(z) = arctan(z)`. Limits a maximum loss on a single residual, has properties similar to 'cauchy'.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Method 'lm' supports only 'linear' loss.

f_scale : float, optional
Value of soft margin between inlier and outlier residuals, default is 1.0. The loss function is evaluated as follows `rho_(f**2) = C**2 * rho(f**2 / C**2)`, where `C` is *f_scale*, and `rho` is determined by *loss* parameter. This parameter has no effect with `loss='linear'`, but for other *loss* values it is of crucial importance.

max_nfev : None or int, optional
Maximum number of function evaluations before the termination. If None (default), the value is chosen automatically:

- For 'trf' and 'dogbox' : 100 * n.
- For 'lm' : 100 * n if *jac* is callable and 100 * n * (n + 1) otherwise (because 'lm' counts function calls in Jacobian estimation).

diff_step : None or array_like, optional

Determines the relative step size for the finite difference approximation of the Jacobian. The actual step is computed as `x * diff_step`. If None (default), then *diff_step* is taken to be a conventional "optimal" power of machine epsilon for the finite difference scheme used *[NR]*.

tr_solver : {None, 'exact', 'lsmr'}, optional

Method for solving trust-region subproblems, relevant only for 'trf' and 'dogbox' methods.

- 'exact' is suitable for not very large problems with dense Jacobian matrices. The computational complexity per iteration is comparable to a singular value decomposition of the Jacobian matrix.
- 'lsmr' is suitable for problems with sparse and large Jacobian matrices. It uses the iterative procedure *`scipy.sparse.linalg.lsmr`* for finding a solution of a linear least-squares problem and only requires matrix-vector product evaluations.

If None (default) the solver is chosen based on the type of Jacobian returned on the first iteration.

tr_options : dict, optional

Keyword options passed to trust-region solver.

- `tr_solver='exact'`: *tr_options* are ignored.
- `tr_solver='lsmr'`: options for *`scipy.sparse.linalg.lsmr`*. Additionally `method='trf'` supports 'regularize' option (bool, default is True) which adds a regularization term to the normal equation, which improves convergence if the Jacobian is rank-deficient *[Byrd]* (eq. 3.4).

jac_sparsity : {None, array_like, sparse matrix}, optional

Defines the sparsity structure of the Jacobian matrix for finite difference estimation, its shape must be (m, n). If the Jacobian has only few non-zero elements in *each* row, providing the sparsity structure will greatly speed up the computations *[Curtis]*. A zero entry means that a corresponding element in the Jacobian is identically zero. If provided, forces the use of 'lsmr' trust-region solver. If None (default) then dense differencing will be used. Has no effect for 'lm' method.

verbose : {0, 1, 2}, optional

Level of algorithm's verbosity:

- 0 (default) : work silently.
- 1 : display a termination report.
- 2 : display progress during iterations (not supported by 'lm' method).

args, kwargs : tuple and dict, optional

Additional arguments passed to *fun* and *jac*. Both empty by default. The calling signature is `fun(x, *args, **kwargs)` and the same for *jac*.

Returns *`OptimizeResult`* with the following fields defined:

x : ndarray, shape (n,)

Solution found.

cost : float

Value of the cost function at the solution.

fun : ndarray, shape (m,)

Vector of residuals at the solution.

jac : ndarray, sparse matrix or LinearOperator, shape (m, n)

Modified Jacobian matrix at the solution, in the sense that J^T J is a Gauss-Newton approximation of the Hessian of the cost function. The type is the same as the one used by the algorithm.

grad : ndarray, shape (m,)

Gradient of the cost function at the solution.

optimality : float

First-order optimality measure. In unconstrained problems, it is always the uniform norm of the gradient. In constrained problems, it is the quantity which was compared with *gtol* during iterations.

active_mask : ndarray of int, shape (n,)
Each component shows whether a corresponding constraint is active (that is, whether a variable is at the bound):
- 0 : a constraint is not active.
- -1 : a lower bound is active.
- 1 : an upper bound is active.

Might be somewhat arbitrary for 'trf' method as it generates a sequence of strictly feasible iterates and *active_mask* is determined within a tolerance threshold.

nfev : int
Number of function evaluations done. Methods 'trf' and 'dogbox' do not count function calls for numerical Jacobian approximation, as opposed to 'lm' method.

njev : int or None
Number of Jacobian evaluations done. If numerical Jacobian approximation is used in 'lm' method, it is set to None.

status : int
The reason for algorithm termination:
- -1 : improper input parameters status returned from MINPACK.
- 0 : the maximum number of function evaluations is exceeded.
- 1 : *gtol* termination condition is satisfied.
- 2 : *ftol* termination condition is satisfied.
- 3 : *xtol* termination condition is satisfied.
- 4 : Both *ftol* and *xtol* termination conditions are satisfied.

message : str
Verbal description of the termination reason.

success : bool
True if one of the convergence criteria is satisfied (*status* > 0).

See also:

leastsq A legacy wrapper for the MINPACK implementation of the Levenberg-Marquadt algorithm.

curve_fit Least-squares minimization applied to a curve fitting problem.

Notes

Method 'lm' (Levenberg-Marquardt) calls a wrapper over least-squares algorithms implemented in MINPACK (lmder, lmdif). It runs the Levenberg-Marquardt algorithm formulated as a trust-region type algorithm. The implementation is based on paper *[JJMore]*, it is very robust and efficient with a lot of smart tricks. It should be your first choice for unconstrained problems. Note that it doesn't support bounds. Also it doesn't work when m < n.

Method 'trf' (Trust Region Reflective) is motivated by the process of solving a system of equations, which constitute the first-order optimality condition for a bound-constrained minimization problem as formulated in *[STIR]*. The algorithm iteratively solves trust-region subproblems augmented by a special diagonal quadratic term and with trust-region shape determined by the distance from the bounds and the direction of the gradient. This enhancements help to avoid making steps directly into bounds and efficiently explore the whole space of variables. To further improve convergence, the algorithm considers search directions reflected from the bounds. To obey theoretical requirements, the algorithm keeps iterates strictly feasible. With dense Jacobians trust-region subproblems are solved by an exact method very similar to the one described in *[JJMore]* (and implemented in MINPACK). The difference from the MINPACK implementation is that a singular value decomposition of a Jacobian matrix is done once per iteration, instead of a QR decomposition and series of Givens rotation eliminations. For large sparse Jacobians a 2-d subspace approach of solving trust-region subproblems is used *[STIR]*, *[Byrd]*. The subspace is spanned by a scaled gradient and an approximate Gauss-Newton solution delivered by `scipy.sparse.linalg.lsmr`. When no constraints are imposed the algorithm is very similar to MINPACK and has generally comparable performance. The algorithm works quite robust in unbounded and bounded problems, thus it is chosen as a default algorithm.

Method 'dogbox' operates in a trust-region framework, but considers rectangular trust regions as opposed to conventional ellipsoids *[Voglis]*. The intersection of a current trust region and initial bounds is again rectangular, so on each iteration a quadratic minimization problem subject to bound constraints is solved approximately by Powell's dogleg method *[NumOpt]*. The required Gauss-Newton step can be computed exactly for dense Jacobians or approximately by *scipy.sparse.linalg.lsmr* for large sparse Jacobians. The algorithm is likely to exhibit slow convergence when the rank of Jacobian is less than the number of variables. The algorithm often outperforms 'trf' in bounded problems with a small number of variables.

Robust loss functions are implemented as described in *[BA]*. The idea is to modify a residual vector and a Jacobian matrix on each iteration such that computed gradient and Gauss-Newton Hessian approximation match the true gradient and Hessian approximation of the cost function. Then the algorithm proceeds in a normal way, i.e. robust loss functions are implemented as a simple wrapper over standard least-squares algorithms.

New in version 0.17.0.

References

[STIR], *[NR]*, *[Byrd]*, *[Curtis]*, *[JJMore]*, *[Voglis]*, *[NumOpt]*, *[BA]*

Examples

In this example we find a minimum of the Rosenbrock function without bounds on independed variables.

```
>>> def fun_rosenbrock(x):
...     return np.array([10 * (x[1] - x[0]**2), (1 - x[0])])
```

Notice that we only provide the vector of the residuals. The algorithm constructs the cost function as a sum of squares of the residuals, which gives the Rosenbrock function. The exact minimum is at `x = [1.0, 1.0]`.

```
>>> from scipy.optimize import least_squares
>>> x0_rosenbrock = np.array([2, 2])
>>> res_1 = least_squares(fun_rosenbrock, x0_rosenbrock)
>>> res_1.x
array([ 1.,  1.])
>>> res_1.cost
9.8669242910846867e-30
>>> res_1.optimality
8.8928864934219529e-14
```

We now constrain the variables, in such a way that the previous solution becomes infeasible. Specifically, we require that `x[1] >= 1.5`, and `x[0]` left unconstrained. To this end, we specify the *bounds* parameter to *least_squares* in the form `bounds=([-np.inf, 1.5], np.inf)`.

We also provide the analytic Jacobian:

```
>>> def jac_rosenbrock(x):
...     return np.array([
...         [-20 * x[0], 10],
...         [-1, 0]])
```

Putting this all together, we see that the new solution lies on the bound:

```
>>> res_2 = least_squares(fun_rosenbrock, x0_rosenbrock, jac_rosenbrock,
...                       bounds=([-np.inf, 1.5], np.inf))
>>> res_2.x
array([ 1.22437075,  1.5       ])
>>> res_2.cost
0.025213093946805685
```

```
>>> res_2.optimality
1.5885401433157753e-07
```

Now we solve a system of equations (i.e., the cost function should be zero at a minimum) for a Broyden tridiagonal vector-valued function of 100000 variables:

```
>>> def fun_broyden(x):
...     f = (3 - x) * x + 1
...     f[1:] -= x[:-1]
...     f[:-1] -= 2 * x[1:]
...     return f
```

The corresponding Jacobian matrix is sparse. We tell the algorithm to estimate it by finite differences and provide the sparsity structure of Jacobian to significantly speed up this process.

```
>>> from scipy.sparse import lil_matrix
>>> def sparsity_broyden(n):
...     sparsity = lil_matrix((n, n), dtype=int)
...     i = np.arange(n)
...     sparsity[i, i] = 1
...     i = np.arange(1, n)
...     sparsity[i, i - 1] = 1
...     i = np.arange(n - 1)
...     sparsity[i, i + 1] = 1
...     return sparsity
...
>>> n = 100000
>>> x0_broyden = -np.ones(n)
...
>>> res_3 = least_squares(fun_broyden, x0_broyden,
...                       jac_sparsity=sparsity_broyden(n))
>>> res_3.cost
4.5687069299604613e-23
>>> res_3.optimality
1.1650454296851518e-11
```

Let's also solve a curve fitting problem using robust loss function to take care of outliers in the data. Define the model function as `y = a + b * exp(c * t)`, where t is a predictor variable, y is an observation and a, b, c are parameters to estimate.

First, define the function which generates the data with noise and outliers, define the model parameters, and generate data:

```
>>> def gen_data(t, a, b, c, noise=0, n_outliers=0, random_state=0):
...     y = a + b * np.exp(t * c)
...
...     rnd = np.random.RandomState(random_state)
...     error = noise * rnd.randn(t.size)
...     outliers = rnd.randint(0, t.size, n_outliers)
...     error[outliers] *= 10
...
...     return y + error
...
>>> a = 0.5
>>> b = 2.0
>>> c = -1
>>> t_min = 0
>>> t_max = 10
```

```
>>> n_points = 15
...
>>> t_train = np.linspace(t_min, t_max, n_points)
>>> y_train = gen_data(t_train, a, b, c, noise=0.1, n_outliers=3)
```

Define function for computing residuals and initial estimate of parameters.

```
>>> def fun(x, t, y):
...     return x[0] + x[1] * np.exp(x[2] * t) - y
...
>>> x0 = np.array([1.0, 1.0, 0.0])
```

Compute a standard least-squares solution:

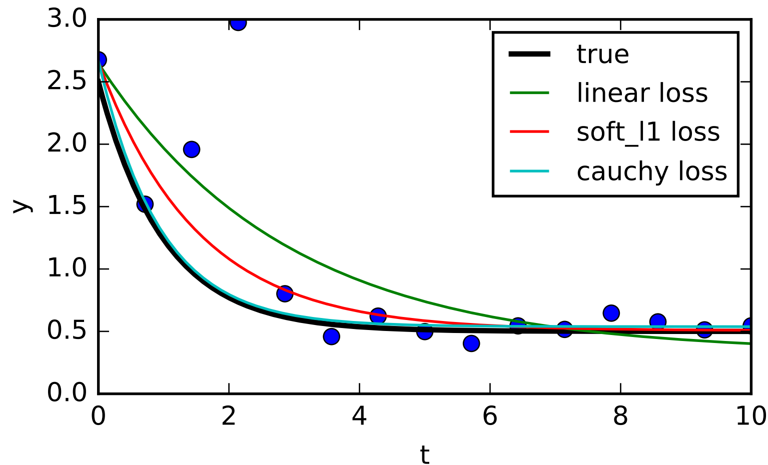
```
>>> res_lsq = least_squares(fun, x0, args=(t_train, y_train))
```

Now compute two solutions with two different robust loss functions. The parameter *f_scale* is set to 0.1, meaning that inlier residuals should not significantly exceed 0.1 (the noise level used).

```
>>> res_soft_l1 = least_squares(fun, x0, loss='soft_l1', f_scale=0.1,
...                             args=(t_train, y_train))
>>> res_log = least_squares(fun, x0, loss='cauchy', f_scale=0.1,
...                         args=(t_train, y_train))
```

And finally plot all the curves. We see that by selecting an appropriate *loss* we can get estimates close to optimal even in the presence of strong outliers. But keep in mind that generally it is recommended to try 'soft_l1' or 'huber' losses first (if at all necessary) as the other two options may cause difficulties in optimization process.

```
>>> t_test = np.linspace(t_min, t_max, n_points * 10)
>>> y_true = gen_data(t_test, a, b, c)
>>> y_lsq = gen_data(t_test, *res_lsq.x)
>>> y_soft_l1 = gen_data(t_test, *res_soft_l1.x)
>>> y_log = gen_data(t_test, *res_log.x)
...
>>> import matplotlib.pyplot as plt
>>> plt.plot(t_train, y_train, 'o')
>>> plt.plot(t_test, y_true, 'k', linewidth=2, label='true')
>>> plt.plot(t_test, y_lsq, label='linear loss')
>>> plt.plot(t_test, y_soft_l1, label='soft_l1 loss')
>>> plt.plot(t_test, y_log, label='cauchy loss')
>>> plt.xlabel("t")
>>> plt.ylabel("y")
>>> plt.legend()
>>> plt.show()
```



In the next example, we show how complex-valued residual functions of complex variables can be optimized with `least_squares()`. Consider the following function:

```
>>> def f(z):
...     return z - (0.5 + 0.5j)
```

We wrap it into a function of real variables that returns real residuals by simply handling the real and imaginary parts as independent variables:

```
>>> def f_wrap(x):
...     fx = f(x[0] + 1j*x[1])
...     return np.array([fx.real, fx.imag])
```

Thus, instead of the original m-dimensional complex function of n complex variables we optimize a 2m-dimensional real function of 2n real variables:

```
>>> from scipy.optimize import least_squares
>>> res_wrapped = least_squares(f_wrap, (0.1, 0.1), bounds=([0, 0], [1, 1]))
>>> z = res_wrapped.x[0] + res_wrapped.x[1]*1j
>>> z
(0.49999999999925893+0.49999999999925893j)
```

`scipy.optimize.`**`nnls`**(*A*, *b*)

Solve `argmin_x || Ax - b ||_2` for `x>=0`. This is a wrapper for a FORTRAN non-negative least squares solver.

Parameters **A** : ndarray
  Matrix `A` as shown above.
b : ndarray
  Right-hand side vector.

Returns **x** : ndarray
  Solution vector.
rnorm : float
  The residual, `|| Ax-b ||_2`.

Notes

The FORTRAN code was published in the book below. The algorithm is an active set method. It solves the KKT (Karush-Kuhn-Tucker) conditions for the non-negative least squares problem.

References

Lawson C., Hanson R.J., (1987) Solving Least Squares Problems, SIAM

`scipy.optimize.`**`lsq_linear`**(*A*, *b*, *bounds=(-inf, inf)*, *method='trf'*, *tol=1e-10*, *lsq_solver=None*, *lsmr_tol=None*, *max_iter=None*, *verbose=0*)

Solve a linear least-squares problem with bounds on the variables.

Given a m-by-n design matrix A and a target vector b with m elements, *lsq_linear* solves the following optimization problem:

```
minimize 0.5 * ||A x - b||**2
subject to lb <= x <= ub
```

This optimization problem is convex, hence a found minimum (if iterations have converged) is guaranteed to be global.

Parameters

A : array_like, sparse matrix of LinearOperator, shape (m, n)
  Design matrix. Can be *scipy.sparse.linalg.LinearOperator*.

b : array_like, shape (m,)
  Target vector.

bounds : 2-tuple of array_like, optional
  Lower and upper bounds on independent variables. Defaults to no bounds. Each array must have shape (n,) or be a scalar, in the latter case a bound will be the same for all variables. Use `np.inf` with an appropriate sign to disable bounds on all or some variables.

method : 'trf' or 'bvls', optional
  Method to perform minimization.
  - 'trf' : Trust Region Reflective algorithm adapted for a linear least-squares problem. This is an interior-point-like method and the required number of iterations is weakly correlated with the number of variables.
  - 'bvls' : Bounded-Variable Least-Squares algorithm. This is an active set method, which requires the number of iterations comparable to the number of variables. Can't be used when *A* is sparse or LinearOperator.

  Default is 'trf'.

tol : float, optional
  Tolerance parameter. The algorithm terminates if a relative change of the cost function is less than *tol* on the last iteration. Additionally the first-order optimality measure is considered:
  - `method='trf'` terminates if the uniform norm of the gradient, scaled to account for the presence of the bounds, is less than *tol*.
  - `method='bvls'` terminates if Karush-Kuhn-Tucker conditions are satisfied within *tol* tolerance.

lsq_solver : {None, 'exact', 'lsmr'}, optional
  Method of solving unbounded least-squares problems throughout iterations:
  - 'exact' : Use dense QR or SVD decomposition approach. Can't be used when *A* is sparse or LinearOperator.
  - 'lsmr' : Use *scipy.sparse.linalg.lsmr* iterative procedure which requires only matrix-vector product evaluations. Can't be used with `method='bvls'`.

  If None (default) the solver is chosen based on type of *A*.

lsmr_tol : None, float or 'auto', optional

Tolerance parameters 'atol' and 'btol' for `scipy.sparse.linalg.lsmr` If None (default), it is set to `1e-2 * tol`. If 'auto', the tolerance will be adjusted based on the optimality of the current iterate, which can speed up the optimization process, but is not always reliable.

max_iter : None or int, optional
Maximum number of iterations before termination. If None (default), it is set to 100 for `method='trf'` or to the number of variables for `method='bvls'` (not counting iterations for 'bvls' initialization).

verbose : {0, 1, 2}, optional
Level of algorithm's verbosity:

- 0 : work silently (default).
- 1 : display a termination report.
- 2 : display progress during iterations.

Returns OptimizeResult with the following fields defined:

x : ndarray, shape (n,)
Solution found.

cost : float
Value of the cost function at the solution.

fun : ndarray, shape (m,)
Vector of residuals at the solution.

optimality : float
First-order optimality measure. The exact meaning depends on *method*, refer to the description of *tol* parameter.

active_mask : ndarray of int, shape (n,)
Each component shows whether a corresponding constraint is active (that is, whether a variable is at the bound):

- 0 : a constraint is not active.
- -1 : a lower bound is active.
- 1 : an upper bound is active.

Might be somewhat arbitrary for the *trf* method as it generates a sequence of strictly feasible iterates and active_mask is determined within a tolerance threshold.

nit : int
Number of iterations. Zero if the unconstrained solution is optimal.

status : int
Reason for algorithm termination:

- -1 : the algorithm was not able to make progress on the last iteration.
- 0 : the maximum number of iterations is exceeded.
- 1 : the first-order optimality measure is less than *tol*.
- 2 : the relative change of the cost function is less than *tol*.
- 3 : the unconstrained solution is optimal.

message : str
Verbal description of the termination reason.

success : bool
True if one of the convergence criteria is satisfied (*status* > 0).

See also:

`nnls` Linear least squares with non-negativity constraint.

`least_squares`
Nonlinear least squares with bounds on the variables.

Notes

The algorithm first computes the unconstrained least-squares solution by `numpy.linalg.lstsq` or `scipy.sparse.linalg.lsmr` depending on *lsq_solver*. This solution is returned as optimal if it lies within the bounds.

Method 'trf' runs the adaptation of the algorithm described in *[STIR]* for a linear least-squares problem. The iterations are essentially the same as in the nonlinear least-squares algorithm, but as the quadratic function model is always accurate, we don't need to track or modify the radius of a trust region. The line search (backtracking) is used as a safety net when a selected step does not decrease the cost function. Read more detailed description of the algorithm in `scipy.optimize.least_squares`.

Method 'bvls' runs a Python implementation of the algorithm described in *[BVLS]*. The algorithm maintains active and free sets of variables, on each iteration chooses a new variable to move from the active set to the free set and then solves the unconstrained least-squares problem on free variables. This algorithm is guaranteed to give an accurate solution eventually, but may require up to n iterations for a problem with n variables. Additionally, an ad-hoc initialization procedure is implemented, that determines which variables to set free or active initially. It takes some number of iterations before actual BVLS starts, but can significantly reduce the number of further iterations.

References

[STIR], *[BVLS]*

Examples

In this example a problem with a large sparse matrix and bounds on the variables is solved.

```
>>> from scipy.sparse import rand
>>> from scipy.optimize import lsq_linear
...
>>> np.random.seed(0)
...
>>> m = 20000
>>> n = 10000
...
>>> A = rand(m, n, density=1e-4)
>>> b = np.random.randn(m)
...
>>> lb = np.random.randn(n)
>>> ub = lb + 1
...
>>> res = lsq_linear(A, b, bounds=(lb, ub), lsmr_tol='auto', verbose=1)
# may vary
The relative change of the cost function is less than `tol`.
Number of iterations 16, initial cost 1.5039e+04, final cost 1.1112e+04,
first-order optimality 4.66e-08.
```

Global Optimization

basinhopping(func, x0[, niter, T, stepsize, ...])	Find the global minimum of a function using the basin-hopping algorithm
brute(func, ranges[, args, Ns, full_output, ...])	Minimize a function over a given range by brute force.
differential_evolution(func, bounds[, args, ...])	Finds the global minimum of a multivariate function.

`scipy.optimize.basinhopping`(*func*, *x0*, *niter=100*, *T=1.0*, *stepsize=0.5*, *minimizer_kwargs=None*, *take_step=None*, *accept_test=None*, *callback=None*, *interval=50*, *disp=False*, *niter_success=None*, *seed=None*)

Find the global minimum of a function using the basin-hopping algorithm

Parameters

func : callable `f(x, *args)`
Function to be optimized. `args` can be passed as an optional item in the dict `minimizer_kwargs`

x0 : ndarray
Initial guess.

niter : integer, optional
The number of basin hopping iterations

T : float, optional
The "temperature" parameter for the accept or reject criterion. Higher "temperatures" mean that larger jumps in function value will be accepted. For best results `T` should be comparable to the separation (in function value) between local minima.

stepsize : float, optional
initial step size for use in the random displacement.

minimizer_kwargs : dict, optional
Extra keyword arguments to be passed to the minimizer `scipy.optimize.minimize()` Some important options could be:

method [str] The minimization method (e.g. `"L-BFGS-B"`)

args [tuple] Extra arguments passed to the objective function (`func`) and its derivatives (Jacobian, Hessian).

take_step : callable `take_step(x)`, optional
Replace the default step taking routine with this routine. The default step taking routine is a random displacement of the coordinates, but other step taking algorithms may be better for some systems. `take_step` can optionally have the attribute `take_step.stepsize`. If this attribute exists, then `basinhopping` will adjust `take_step.stepsize` in order to try to optimize the global minimum search.

accept_test : callable, `accept_test(f_new=f_new, x_new=x_new, f_old=fold, x_old=x_old)`, optional
Define a test which will be used to judge whether or not to accept the step. This will be used in addition to the Metropolis test based on "temperature" `T`. The acceptable return values are True, False, or `"force accept"`. If any of the tests return False then the step is rejected. If the latter, then this will override any other tests in order to accept the step. This can be used, for example, to forcefully escape from a local minimum that `basinhopping` is trapped in.

callback : callable, `callback(x, f, accept)`, optional
A callback function which will be called for all minima found. `x` and `f` are the coordinates and function value of the trial minimum, and `accept` is whether or not that minimum was accepted. This can be used, for example, to save the lowest N minima found. Also, `callback` can be used to specify a user defined stop criterion by optionally returning True to stop the `basinhopping` routine.

interval : integer, optional
interval for how often to update the `stepsize`

disp : bool, optional
Set to True to print status messages

niter_success : integer, optional
Stop the run if the global minimum candidate remains the same for this number of iterations.

seed : int or *np.random.RandomState*, optional
If *seed* is not specified the *np.RandomState* singleton is used. If *seed* is an int, a new *np.random.RandomState* instance is used, seeded with seed. If *seed* is already a *np.random.RandomState instance*, then that *np.random.RandomState* instance is used.

Specify *seed* for repeatable minimizations. The random numbers generated with this seed only affect the default Metropolis *accept_test* and the default *take_step*. If you supply your own *take_step* and *accept_test*, and these functions use random number generation, then those functions are responsible for the state of their random number generator.

Returns **res** : OptimizeResult

The optimization result represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `fun` the value of the function at the solution, and `message` which describes the cause of the termination. The `OptimzeResult` object returned by the selected minimizer at the lowest minimum is also contained within this object and can be accessed through the `lowest_optimization_result` attribute. See *`OptimizeResult`* for a description of other attributes.

See also:

`minimize` The local minimization function called once for each basinhopping step. `minimizer_kwargs` is passed to this routine.

Notes

Basin-hopping is a stochastic algorithm which attempts to find the global minimum of a smooth scalar function of one or more variables *[R158] [R159] [R160] [R161]*. The algorithm in its current form was described by David Wales and Jonathan Doye *[R159]* http://www-wales.ch.cam.ac.uk/.

The algorithm is iterative with each cycle composed of the following features

1. random perturbation of the coordinates
2. local minimization
3. accept or reject the new coordinates based on the minimized function value

The acceptance test used here is the Metropolis criterion of standard Monte Carlo algorithms, although there are many other possibilities *[R160]*.

This global minimization method has been shown to be extremely efficient for a wide variety of problems in physics and chemistry. It is particularly useful when the function has many minima separated by large barriers. See the Cambridge Cluster Database http://www-wales.ch.cam.ac.uk/CCD.html for databases of molecular systems that have been optimized primarily using basin-hopping. This database includes minimization problems exceeding 300 degrees of freedom.

See the free software program GMIN (http://www-wales.ch.cam.ac.uk/GMIN) for a Fortran implementation of basin-hopping. This implementation has many different variations of the procedure described above, including more advanced step taking algorithms and alternate acceptance criterion.

For stochastic global optimization there is no way to determine if the true global minimum has actually been found. Instead, as a consistency check, the algorithm can be run from a number of different random starting points to ensure the lowest minimum found in each example has converged to the global minimum. For this reason `basinhopping` will by default simply run for the number of iterations `niter` and return the lowest minimum found. It is left to the user to ensure that this is in fact the global minimum.

Choosing `stepsize`: This is a crucial parameter in `basinhopping` and depends on the problem being solved. Ideally it should be comparable to the typical separation between local minima of the function being optimized. `basinhopping` will, by default, adjust `stepsize` to find an optimal value, but this may take many iterations. You will get quicker results if you set a sensible value for `stepsize`.

Choosing `T`: The parameter `T` is the temperature used in the metropolis criterion. Basinhopping steps are accepted with probability `1` if `func(xnew) < func(xold)`, or otherwise with probability:

```
exp( -(func(xnew) - func(xold)) / T )
```

So, for best results, `T` should to be comparable to the typical difference in function values between local minima.

New in version 0.12.0.

References

[R158], [R159], [R160], [R161]

Examples

The following example is a one-dimensional minimization problem, with many local minima superimposed on a parabola.

```
>>> from scipy.optimize import basinhopping
>>> func = lambda x: np.cos(14.5 * x - 0.3) + (x + 0.2) * x
>>> x0=[1.]
```

Basinhopping, internally, uses a local minimization algorithm. We will use the parameter `minimizer_kwargs` to tell basinhopping which algorithm to use and how to set up that minimizer. This parameter will be passed to `scipy.optimize.minimize()`.

```
>>> minimizer_kwargs = {"method": "BFGS"}
>>> ret = basinhopping(func, x0, minimizer_kwargs=minimizer_kwargs,
...                    niter=200)
>>> print("global minimum: x = %.4f, f(x0) = %.4f" % (ret.x, ret.fun))
global minimum: x = -0.1951, f(x0) = -1.0009
```

Next consider a two-dimensional minimization problem. Also, this time we will use gradient information to significantly speed up the search.

```
>>> def func2d(x):
...     f = np.cos(14.5 * x[0] - 0.3) + (x[1] + 0.2) * x[1] + (x[0] +
...                                                          0.2) * x[0]
...     df = np.zeros(2)
...     df[0] = -14.5 * np.sin(14.5 * x[0] - 0.3) + 2. * x[0] + 0.2
...     df[1] = 2. * x[1] + 0.2
...     return f, df
```

We'll also use a different local minimization algorithm. Also we must tell the minimizer that our function returns both energy and gradient (jacobian)

```
>>> minimizer_kwargs = {"method":"L-BFGS-B", "jac":True}
>>> x0 = [1.0, 1.0]
>>> ret = basinhopping(func2d, x0, minimizer_kwargs=minimizer_kwargs,
...                    niter=200)
>>> print("global minimum: x = [%.4f, %.4f], f(x0) = %.4f" % (ret.x[0],
...                                                           ret.x[1],
...                                                           ret.fun))
global minimum: x = [-0.1951, -0.1000], f(x0) = -1.0109
```

Here is an example using a custom step taking routine. Imagine you want the first coordinate to take larger steps then the rest of the coordinates. This can be implemented like so:

```
>>> class MyTakeStep(object):
...    def __init__(self, stepsize=0.5):
...        self.stepsize = stepsize
```

```
...     def __call__(self, x):
...         s = self.stepsize
...         x[0] += np.random.uniform(-2.*s, 2.*s)
...         x[1:] += np.random.uniform(-s, s, x[1:].shape)
...         return x
```

Since `MyTakeStep.stepsize` exists basinhopping will adjust the magnitude of `stepsize` to optimize the search. We'll use the same 2-D function as before

```
>>> mytakestep = MyTakeStep()
>>> ret = basinhopping(func2d, x0, minimizer_kwargs=minimizer_kwargs,
...                    niter=200, take_step=mytakestep)
>>> print("global minimum: x = [%.4f, %.4f], f(x0) = %.4f" % (ret.x[0],
...                                                           ret.x[1],
...                                                           ret.fun))
global minimum: x = [-0.1951, -0.1000], f(x0) = -1.0109
```

Now let's do an example using a custom callback function which prints the value of every minimum found

```
>>> def print_fun(x, f, accepted):
...         print("at minimum %.4f accepted %d" % (f, int(accepted)))
```

We'll run it for only 10 basinhopping steps this time.

```
>>> np.random.seed(1)
>>> ret = basinhopping(func2d, x0, minimizer_kwargs=minimizer_kwargs,
...                    niter=10, callback=print_fun)
at minimum 0.4159 accepted 1
at minimum -0.9073 accepted 1
at minimum -0.1021 accepted 1
at minimum -0.1021 accepted 1
at minimum 0.9102 accepted 1
at minimum 0.9102 accepted 1
at minimum 2.2945 accepted 0
at minimum -0.1021 accepted 1
at minimum -1.0109 accepted 1
at minimum -1.0109 accepted 1
```

The minimum at -1.0109 is actually the global minimum, found already on the 8th iteration.

Now let's implement bounds on the problem using a custom `accept_test`:

```
>>> class MyBounds(object):
...     def __init__(self, xmax=[1.1,1.1], xmin=[-1.1,-1.1] ):
...         self.xmax = np.array(xmax)
...         self.xmin = np.array(xmin)
...     def __call__(self, **kwargs):
...         x = kwargs["x_new"]
...         tmax = bool(np.all(x <= self.xmax))
...         tmin = bool(np.all(x >= self.xmin))
...         return tmax and tmin
```

```
>>> mybounds = MyBounds()
>>> ret = basinhopping(func2d, x0, minimizer_kwargs=minimizer_kwargs,
...                    niter=10, accept_test=mybounds)
```

`scipy.optimize.`**`brute`**(*func*, *ranges*, *args=()*, *Ns=20*, *full_output=0*, *finish=<function fmin>*, *disp=False*)

Minimize a function over a given range by brute force.

Uses the "brute force" method, i.e. computes the function's value at each point of a multidimensional grid of points, to find the global minimum of the function.

The function is evaluated everywhere in the range with the datatype of the first call to the function, as enforced by the `vectorize` NumPy function. The value and type of the function evaluation returned when `full_output=True` are affected in addition by the `finish` argument (see Notes).

Parameters **func** : callable

The objective function to be minimized. Must be in the form `f(x, *args)`, where `x` is the argument in the form of a 1-D array and `args` is a tuple of any additional fixed parameters needed to completely specify the function.

ranges : tuple

Each component of the *ranges* tuple must be either a "slice object" or a range tuple of the form `(low, high)`. The program uses these to create the grid of points on which the objective function will be computed. See *Note 2* for more detail.

args : tuple, optional

Any additional fixed parameters needed to completely specify the function.

Ns : int, optional

Number of grid points along the axes, if not otherwise specified. See *Note2*.

full_output : bool, optional

If True, return the evaluation grid and the objective function's values on it.

finish : callable, optional

An optimization function that is called with the result of brute force minimization as initial guess. *finish* should take *func* and the initial guess as positional arguments, and take `args` as keyword arguments. It may additionally take *full_output* and/or *disp* as keyword arguments. Use None if no "polishing" function is to be used. See Notes for more details.

disp : bool, optional

Set to True to print convergence messages.

Returns **x0** : ndarray

A 1-D array containing the coordinates of a point at which the objective function had its minimum value. (See *Note 1* for which point is returned.)

fval : float

Function value at the point *x0*. (Returned when *full_output* is True.)

grid : tuple

Representation of the evaluation grid. It has the same length as *x0*. (Returned when *full_output* is True.)

Jout : ndarray

Function values at each point of the evaluation grid, *i.e.*, `Jout = func(*grid)`. (Returned when *full_output* is True.)

See also:

`basinhopping`, *`differential_evolution`*

Notes

Note 1: The program finds the gridpoint at which the lowest value of the objective function occurs. If *finish* is None, that is the point returned. When the global minimum occurs within (or not very far outside) the grid's boundaries, and the grid is fine enough, that point will be in the neighborhood of the global minimum.

However, users often employ some other optimization program to "polish" the gridpoint values, *i.e.*, to seek a more precise (local) minimum near *brute's* best gridpoint. The `brute` function's *finish* option provides a convenient way to do that. Any polishing program used must take *brute's* output as its initial guess as a

positional argument, and take *brute's* input values for `args` as keyword arguments, otherwise an error will be raised. It may additionally take *full_output* and/or *disp* as keyword arguments.

`brute` assumes that the *finish* function returns either an `OptimizeResult` object or a tuple in the form: `(xmin, Jmin, ... , statuscode)`, where `xmin` is the minimizing value of the argument, `Jmin` is the minimum value of the objective function, "..." may be some other returned values (which are not used by `brute`), and `statuscode` is the status code of the *finish* program.

Note that when *finish* is not None, the values returned are those of the *finish* program, *not* the gridpoint ones. Consequently, while `brute` confines its search to the input grid points, the *finish* program's results usually will not coincide with any gridpoint, and may fall outside the grid's boundary. Thus, if a minimum only needs to be found over the provided grid points, make sure to pass in *finish=None*.

Note 2: The grid of points is a `numpy.mgrid` object. For `brute` the *ranges* and *Ns* inputs have the following effect. Each component of the *ranges* tuple can be either a slice object or a two-tuple giving a range of values, such as (0, 5). If the component is a slice object, `brute` uses it directly. If the component is a two-tuple range, `brute` internally converts it to a slice object that interpolates *Ns* points from its low-value to its high-value, inclusive.

Examples

We illustrate the use of `brute` to seek the global minimum of a function of two variables that is given as the sum of a positive-definite quadratic and two deep "Gaussian-shaped" craters. Specifically, define the objective function *f* as the sum of three other functions, `f = f1 + f2 + f3`. We suppose each of these has a signature `(z, *params)`, where `z = (x, y)`, and `params` and the functions are as defined below.

```
>>> params = (2, 3, 7, 8, 9, 10, 44, -1, 2, 26, 1, -2, 0.5)
>>> def f1(z, *params):
...     x, y = z
...     a, b, c, d, e, f, g, h, i, j, k, l, scale = params
...     return (a * x**2 + b * x * y + c * y**2 + d*x + e*y + f)
```

```
>>> def f2(z, *params):
...     x, y = z
...     a, b, c, d, e, f, g, h, i, j, k, l, scale = params
...     return (-g*np.exp(-((x-h)**2 + (y-i)**2) / scale))
```

```
>>> def f3(z, *params):
...     x, y = z
...     a, b, c, d, e, f, g, h, i, j, k, l, scale = params
...     return (-j*np.exp(-((x-k)**2 + (y-l)**2) / scale))
```

```
>>> def f(z, *params):
...     return f1(z, *params) + f2(z, *params) + f3(z, *params)
```

Thus, the objective function may have local minima near the minimum of each of the three functions of which it is composed. To use `fmin` to polish its gridpoint result, we may then continue as follows:

```
>>> rranges = (slice(-4, 4, 0.25), slice(-4, 4, 0.25))
>>> from scipy import optimize
>>> resbrute = optimize.brute(f, rranges, args=params, full_output=True,
...                           finish=optimize.fmin)
>>> resbrute[0]  # global minimum
array([-1.05665192,  1.80834843])
>>> resbrute[1]  # function value at global minimum
-3.4085818767
```

Note that if *finish* had been set to None, we would have gotten the gridpoint [-1.0 1.75] where the rounded function value is -2.892.

Rosenbrock function

rosen(x)	The Rosenbrock function.
rosen_der(x)	The derivative (i.e.
rosen_hess(x)	The Hessian matrix of the Rosenbrock function.
rosen_hess_prod(x, p)	Product of the Hessian matrix of the Rosenbrock function with a vector.

scipy.optimize.**rosen**(*x*)
The Rosenbrock function.

The function computed is:

```
sum(100.0*(x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.0)
```

Parameters **x** : array_like
1-D array of points at which the Rosenbrock function is to be computed.

Returns **f** : float
The value of the Rosenbrock function.

See also:

rosen_der, *rosen_hess*, *rosen_hess_prod*

scipy.optimize.**rosen_der**(*x*)
The derivative (i.e. gradient) of the Rosenbrock function.

Parameters **x** : array_like
1-D array of points at which the derivative is to be computed.

Returns **rosen_der** : (N,) ndarray
The gradient of the Rosenbrock function at *x*.

See also:

rosen, *rosen_hess*, *rosen_hess_prod*

scipy.optimize.**rosen_hess**(*x*)
The Hessian matrix of the Rosenbrock function.

Parameters **x** : array_like
1-D array of points at which the Hessian matrix is to be computed.

Returns **rosen_hess** : ndarray
The Hessian matrix of the Rosenbrock function at *x*.

See also:

rosen, *rosen_der*, *rosen_hess_prod*

scipy.optimize.**rosen_hess_prod**(*x*, *p*)
Product of the Hessian matrix of the Rosenbrock function with a vector.

Parameters **x** : array_like
1-D array of points at which the Hessian matrix is to be computed.

p : array_like

1-D array, the vector to be multiplied by the Hessian matrix.

Returns **rosen_hess_prod** : ndarray
The Hessian matrix of the Rosenbrock function at *x* multiplied by the vector *p*.

See also:

`rosen`, `rosen_der`, `rosen_hess`

5.18.2 Fitting

`curve_fit`(f, xdata, ydata[, p0, sigma, ...])	Use non-linear least squares to fit a function, f, to data.

`scipy.optimize.`**`curve_fit`**(*f*, *xdata*, *ydata*, *p0=None*, *sigma=None*, *absolute_sigma=False*, *check_finite=True*, *bounds=(-inf*, *inf)*, *method=None*, *jac=None*, ***kwargs*)

Use non-linear least squares to fit a function, f, to data.

Assumes `ydata = f(xdata, *params) + eps`

Parameters **f** : callable
The model function, f(x, ...). It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

xdata : An M-length sequence or an (k,M)-shaped array for functions with k predictors
The independent variable where the data is measured.

ydata : M-length sequence
The dependent data — nominally f(xdata, ...)

p0 : None, scalar, or N-length sequence, optional
Initial guess for the parameters. If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a ValueError is raised).

sigma : None or M-length sequence or MxM array, optional
Determines the uncertainty in *ydata*. If we define residuals as `r = ydata - f(xdata, *popt)`, then the interpretation of *sigma* depends on its number of dimensions:

- A 1-d *sigma* should contain values of standard deviations of errors in *ydata*. In this case, the optimized function is `chisq = sum((r / sigma) ** 2)`.
- A 2-d *sigma* should contain the covariance matrix of errors in *ydata*. In this case, the optimized function is `chisq = r.T @ inv(sigma) @ r`.
  New in version 0.19.

None (default) is equivalent of 1-d *sigma* filled with ones.

absolute_sigma : bool, optional
If True, *sigma* is used in an absolute sense and the estimated parameter covariance *pcov* reflects these absolute values.
If False, only the relative magnitudes of the *sigma* values matter. The returned parameter covariance matrix *pcov* is based on scaling *sigma* by a constant factor. This constant is set by demanding that the reduced *chisq* for the optimal parameters *popt* when using the *scaled sigma* equals unity. In other words, *sigma* is scaled to match the sample variance of the residuals after the fit. Mathematically, `pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt)/(M-N)`

check_finite : bool, optional
If True, check that the input arrays do not contain nans of infs, and raise a ValueError if they do. Setting this parameter to False may silently produce nonsensical results if the input arrays do contain nans. Default is True.

bounds : 2-tuple of array_like, optional
Lower and upper bounds on independent variables. Defaults to no bounds. Each element of the tuple must be either an array with the length equal to the number of parameters, or a scalar (in which case the bound is taken to be the same for all parameters.) Use `np.inf` with an appropriate sign to disable bounds on all or some parameters.
New in version 0.17.

method : {'lm', 'trf', 'dogbox'}, optional
Method to use for optimization. See *least_squares* for more details. Default is 'lm' for unconstrained problems and 'trf' if *bounds* are provided. The method 'lm' won't work when the number of observations is less than the number of variables, use 'trf' or 'dogbox' in this case.
New in version 0.17.

jac : callable, string or None, optional
Function with signature `jac(x, ...)` which computes the Jacobian matrix of the model function with respect to parameters as a dense array_like structure. It will be scaled according to provided *sigma*. If None (default), the Jacobian will be estimated numerically. String keywords for 'trf' and 'dogbox' methods can be used to select a finite difference scheme, see *least_squares*.
New in version 0.18.

kwargs
Keyword arguments passed to *leastsq* for `method='lm'` or *least_squares* otherwise.

Returns

popt : array
Optimal values for the parameters so that the sum of the squared residuals of `f(xdata, *popt) - ydata` is minimized

pcov : 2d array
The estimated covariance of popt. The diagonals provide the variance of the parameter estimate. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
How the *sigma* parameter affects the estimated covariance depends on *absolute_sigma* argument, as described above.
If the Jacobian matrix at the solution doesn't have a full rank, then 'lm' method returns a matrix filled with `np.inf`, on the other hand 'trf' and 'dogbox' methods use Moore-Penrose pseudoinverse to compute the covariance matrix.

Raises

ValueError
if either *ydata* or *xdata* contain NaNs, or if incompatible options are used.

RuntimeError
if the least-squares minimization fails.

OptimizeWarning
if covariance of the parameters can not be estimated.

See also:

`least_squares`
Minimize the sum of squares of nonlinear functions.

`scipy.stats.linregress`
Calculate a linear least squares regression for two sets of measurements.

Notes

With `method='lm'`, the algorithm uses the Levenberg-Marquardt algorithm through *leastsq*. Note that this algorithm can only deal with unconstrained problems.

Box constraints can be handled by methods 'trf' and 'dogbox'. Refer to the docstring of *least_squares* for more information.

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.optimize import curve_fit
```

```
>>> def func(x, a, b, c):
...     return a * np.exp(-b * x) + c
```

define the data to be fit with some noise

```
>>> xdata = np.linspace(0, 4, 50)
>>> y = func(xdata, 2.5, 1.3, 0.5)
>>> y_noise = 0.2 * np.random.normal(size=xdata.size)
>>> ydata = y + y_noise
>>> plt.plot(xdata, ydata, 'b-', label='data')
```

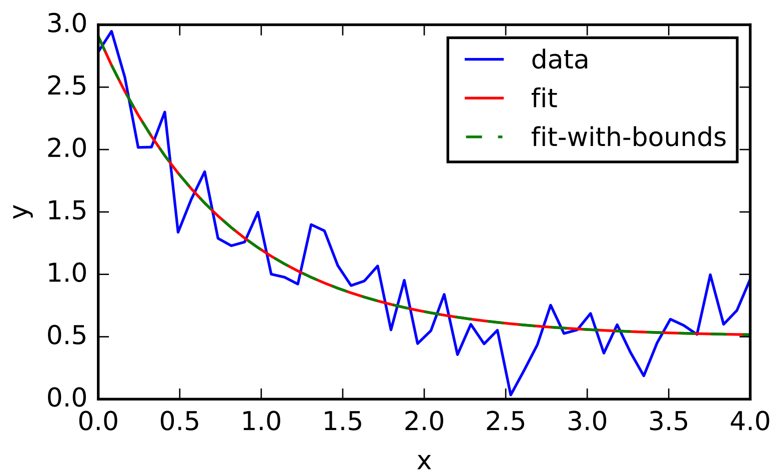
Fit for the parameters a, b, c of the function *func*

```
>>> popt, pcov = curve_fit(func, xdata, ydata)
>>> plt.plot(xdata, func(xdata, *popt), 'r-', label='fit')
```

Constrain the optimization to the region of `0 < a < 3`, `0 < b < 2` and `0 < c < 1`:

```
>>> popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [3., 2., 1.]))
>>> plt.plot(xdata, func(xdata, *popt), 'g--', label='fit-with-bounds')
```

```
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.show()
```



5.18.3 Root finding

Scalar functions

brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of f in [a,b].
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval.
newton(func, x0[, fprime, args, tol, ...])	Find a zero using the Newton-Raphson or secant method.

scipy.optimize.**brentq**(*f*, *a*, *b*, *args=()*, *xtol=2e-12*, *rtol=8.8817841970012523e-16*, *maxiter=100*, *full_output=False*, *disp=True*)

Find a root of a function in a bracketing interval using Brent's method.

Uses the classic Brent's method to find a zero of the function *f* on the sign changing interval [a , b]. Generally considered the best of the rootfinding routines here. It is a safe version of the secant method that uses inverse quadratic extrapolation. Brent's method combines root bracketing, interval bisection, and inverse quadratic interpolation. It is sometimes known as the van Wijngaarden-Dekker-Brent method. Brent (1973) claims convergence is guaranteed for functions computable within [a,b].

[Brent1973] provides the classic description of the algorithm. Another description can be found in a recent edition of Numerical Recipes, including *[PressEtal1992]*. Another description is at http://mathworld.wolfram.com/BrentsMethod.html. It should be easy to understand the algorithm just by reading our code. Our code diverges a bit from standard presentations: we choose a different formula for the extrapolation step.

Parameters

f : function
: Python function returning a number. The function f must be continuous, and $f(a)$ and $f(b)$ must have opposite signs.

a : number
: One end of the bracketing interval $[a, b]$.

b : number
: The other end of the bracketing interval $[a, b]$.

xtol : number, optional
: The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter must be nonnegative. For nice functions, Brent's method will often satisfy the above condition will `xtol/2` and `rtol/2`. *[Brent1973]*

rtol : number, optional
: The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter cannot be smaller than its default value of `4*np.finfo(float).eps`. For nice functions, Brent's method will often satisfy the above condition will `xtol/2` and `rtol/2`. *[Brent1973]*

maxiter : number, optional
: if convergence is not achieved in maxiter iterations, an error is raised. Must be >= 0.

args : tuple, optional
: containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional
: If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : bool, optional
: If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float
: Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)
: Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See also:

multivariate
fmin, *fmin_powell*, *fmin_cg*, *fmin_bfgs*, *fmin_ncg*

nonlinear *leastsq*

constrained
fmin_l_bfgs_b, *fmin_tnc*, *fmin_cobyla*

global *basinhopping*, *brute*, *differential_evolution*

local *fminbound*, *brent*, *golden*, *bracket*

n-dimensional
fsolve

one-dimensional
brenth, *ridder*, *bisect*, *newton*

scalar *fixed_point*

Notes

f must be continuous. f(a) and f(b) must have opposite signs.

References

[Brent1973], *[PressEtal1992]*

`scipy.optimize.`**`brenth`**(*f*, *a*, *b*, *args=()*, *xtol=2e-12*, *rtol=8.8817841970012523e-16*, *maxiter=100*, *full_output=False*, *disp=True*)

Find root of f in [a,b].

A variation on the classic Brent routine to find a zero of the function f between the arguments a and b that uses hyperbolic extrapolation instead of inverse quadratic extrapolation. There was a paper back in the 1980's ... f(a) and f(b) cannot have the same signs. Generally on a par with the brent routine, but not as heavily tested. It is a safe version of the secant method that uses hyperbolic extrapolation. The version here is by Chuck Harris.

Parameters

f : function
Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number
One end of the bracketing interval [a,b].

b : number
The other end of the bracketing interval [a,b].

xtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter must be nonnegative. As with *brentq*, for nice functions the method will often satisfy the above condition will `xtol/2` and `rtol/2`.

rtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter cannot be smaller than its default value of `4*np.finfo(float).eps`. As with *brentq*, for nice functions the method will often satisfy the above condition will `xtol/2` and `rtol/2`.

maxiter : number, optional
if convergence is not achieved in maxiter iterations, an error is raised. Must be >= 0.

args : tuple, optional
containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : bool, optional
If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float
Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)
Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See also:

fmin, *fmin_powell*, *fmin_cg*

leastsq nonlinear least squares minimizer

fmin_l_bfgs_b, *fmin_tnc*, *fmin_cobyla*, *basinhopping*, *differential_evolution*, *brute*, *fminbound*, *brent*, *golden*, *bracket*

fsolve n-dimensional root-finding

brentq, *brenth*, *ridder*, *bisect*, *newton*

fixed_point
scalar fixed-point finder

scipy.optimize.**ridder**(*f, a, b, args=(), xtol=2e-12, rtol=8.8817841970012523e-16, maxiter=100, full_output=False, disp=True*)

Find a root of a function in an interval.

Parameters

f : function
Python function returning a number. f must be continuous, and f(a) and f(b) must have opposite signs.

a : number
One end of the bracketing interval [a,b].

b : number
The other end of the bracketing interval [a,b].

xtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter must be nonnegative.

rtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter cannot be smaller than its default value of `4*np.finfo(float).eps`.

maxiter : number, optional
if convergence is not achieved in maxiter iterations, an error is raised. Must be >= 0.

args : tuple, optional
containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional
If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where *x* is the root, and *r* is a RootResults object.

disp : bool, optional
If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float
Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)
Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See also:

brentq, *brenth*, *bisect*, *newton*

fixed_point
scalar fixed-point finder

Notes

Uses *[Ridders1979]* method to find a zero of the function *f* between the arguments *a* and *b*. Ridders' method is faster than bisection, but not generally as fast as the Brent rountines. *[Ridders1979]* provides the classic description and source of the algorithm. A description can also be found in any recent edition of Numerical Recipes.

The routine used here diverges slightly from standard presentations in order to be a bit more careful of tolerance.

References

[Ridders1979]

scipy.optimize.**bisect**(*f*, *a*, *b*, *args=()*, *xtol=2e-12*, *rtol=8.881784197001252e-16*, *maxiter=100*, *full_output=False*, *disp=True*)

Find root of a function within an interval.

Basic bisection routine to find a zero of the function *f* between the arguments *a* and *b*. *f(a)* and *f(b)* cannot have the same signs. Slow but sure.

Parameters

f : function
Python function returning a number. *f* must be continuous, and f(a) and f(b) must have opposite signs.

a : number
One end of the bracketing interval [a,b].

b : number
The other end of the bracketing interval [a,b].

xtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter must be nonnegative.

rtol : number, optional
The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter cannot be smaller than its default value of `4*np.finfo(float).eps`.

maxiter : number, optional
if convergence is not achieved in *maxiter* iterations, an error is raised. Must be >= 0.

args : tuple, optional
containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

full_output : bool, optional
If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where x is the root, and r is a *RootResults* object.

disp : bool, optional
If True, raise RuntimeError if the algorithm didn't converge.

Returns

x0 : float
Zero of *f* between *a* and *b*.

r : RootResults (present if `full_output = True`)
Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

See also:

brentq, *brenth*, *bisect*, *newton*

fixed_point
scalar fixed-point finder

fsolve n-dimensional root-finding

scipy.optimize.**newton**(*func*, *x0*, *fprime=None*, *args=()*, *tol=1.48e-08*, *maxiter=50*, *fprime2=None*)

Find a zero using the Newton-Raphson or secant method.

Find a zero of the function *func* given a nearby starting point *x0*. The Newton-Raphson method is used if the derivative *fprime* of *func* is provided, otherwise the secant method is used. If the second order derivate *fprime2* of *func* is provided, parabolic Halley's method is used.

Parameters

func : function
The function whose zero is wanted. It must be a function of a single variable of the form f(x,a,b,c...), where a,b,c... are extra arguments that can be passed in the `args` parameter.

x0 : float
An initial estimate of the zero that should be somewhere near the actual zero.

fprime : function, optional
The derivative of the function when available and convenient. If it is None (default), then the secant method is used.

args : tuple, optional
Extra arguments to be used in the function call.

tol : float, optional
The allowable error of the zero value.

maxiter : int, optional
Maximum number of iterations.

fprime2 : function, optional
The second order derivative of the function when available and convenient. If it is None (default), then the normal Newton-Raphson or the secant method is used. If it is given, parabolic Halley's method is used.

Returns

zero : float
Estimated location where function is zero.

See also:

brentq, *brenth*, *ridder*, *bisect*

fsolve find zeroes in n dimensions.

Notes

The convergence rate of the Newton-Raphson method is quadratic, the Halley method is cubic, and the secant method is sub-quadratic. This means that if the function is well behaved the actual error in the estimated zero is approximately the square (cube for Halley) of the requested tolerance up to roundoff error. However, the stopping criterion used here is the step size and there is no guarantee that a zero has been found. Consequently the result should be verified. Safer algorithms are brentq, brenth, ridder, and bisect, but they all require that the root first be bracketed in an interval where the function changes sign. The brentq algorithm is recommended for general use in one dimensional problems when such an interval has been found.

Fixed point finding:

fixed_point(func, x0[, args, xtol, maxiter, ...])	Find a fixed point of the function.

scipy.optimize.**fixed_point**(*func*, *x0*, *args=()*, *xtol=1e-08*, *maxiter=500*, *method='del2'*)

Find a fixed point of the function.

Given a function of one or more variables and a starting point, find a fixed-point of the function: i.e. where

func(x0) == x0.

Parameters

func : function
Function to evaluate.

x0 : array_like
Fixed point of function.

args : tuple, optional
Extra arguments to *func*.

xtol : float, optional
Convergence tolerance, defaults to 1e-08.

maxiter : int, optional
Maximum number of iterations, defaults to 500.

method : {"del2", "iteration"}, optional
Method of finding the fixed-point, defaults to "del2" which uses Steffensen's Method with Aitken's `Del^2` convergence acceleration *[R167]*. The "iteration" method simply iterates the function until convergence is detected, without attempting to accelerate the convergence.

References

[R167]

Examples

```
>>> from scipy import optimize
>>> def func(x, c1, c2):
...     return np.sqrt(c1/(x+c2))
>>> c1 = np.array([10,12.])
>>> c2 = np.array([3, 5.])
>>> optimize.fixed_point(func, [1.2, 1.3], args=(c1,c2))
array([ 1.4920333 ,  1.37228132])
```

Multidimensional

General nonlinear solvers:

root(fun, x0[, args, method, jac, tol, ...])	Find a root of a vector function.
fsolve(func, x0[, args, fprime, ...])	Find the roots of a function.
broyden1(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's first Jacobian approximation.
broyden2(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's second Jacobian approximation.

scipy.optimize.**root**(*fun*, *x0*, *args=()*, *method='hybr'*, *jac=None*, *tol=None*, *callback=None*, *options=None*)

Find a root of a vector function.

Parameters

fun : callable
A vector function to find a root of.

x0 : ndarray
Initial guess.

args : tuple, optional
Extra arguments passed to the objective function and its Jacobian.

method : str, optional
Type of solver. Should be one of

- 'hybr' *(see here)*
- 'lm' *(see here)*
- 'broyden1' *(see here)*
- 'broyden2' *(see here)*
- 'anderson' *(see here)*
- 'linearmixing' *(see here)*
- 'diagbroyden' *(see here)*
- 'excitingmixing' *(see here)*
- 'krylov' *(see here)*
- 'df-sane' *(see here)*

jac : bool or callable, optional
If *jac* is a Boolean and is True, *fun* is assumed to return the value of Jacobian along with the objective function. If False, the Jacobian will be estimated numerically. *jac* can also be a callable returning the Jacobian of *fun*. In this case, it must accept the same arguments as *fun*.

tol : float, optional
Tolerance for termination. For detailed control, use solver-specific options.

callback : function, optional
Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual. For all methods but 'hybr' and 'lm'.

options : dict, optional
A dictionary of solver options. E.g. *xtol* or *maxiter*, see `show_options()` for details.

Returns

sol : OptimizeResult
The solution represented as a `OptimizeResult` object. Important attributes are: `x` the solution array, `success` a Boolean flag indicating if the algorithm exited successfully and `message` which describes the cause of the termination. See `OptimizeResult` for a description of other attributes.

See also:

`show_options`
Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *hybr*.

Method *hybr* uses a modification of the Powell hybrid method as implemented in MINPACK *[R185]*.

Method *lm* solves the system of nonlinear equations in a least squares sense using a modification of the Levenberg-Marquardt algorithm as implemented in MINPACK *[R185]*.

Method *df-sane* is a derivative-free spectral method. *[R187]*

Methods *broyden1*, *broyden2*, *anderson*, *linearmixing*, *diagbroyden*, *excitingmixing*, *krylov* are inexact Newton methods, with backtracking or full line searches *[R186]*. Each method corresponds to a particular Jacobian approximations. See `nonlin` for details.

- Method *broyden1* uses Broyden's first Jacobian approximation, it is known as Broyden's good method.
- Method *broyden2* uses Broyden's second Jacobian approximation, it is known as Broyden's bad method.
- Method *anderson* uses (extended) Anderson mixing.
- Method *Krylov* uses Krylov approximation for inverse Jacobian. It is suitable for large-scale problem.

- Method *diagbroyden* uses diagonal Broyden Jacobian approximation.
- Method *linearmixing* uses a scalar Jacobian approximation.
- Method *excitingmixing* uses a tuned diagonal Jacobian approximation.

Warning: The algorithms implemented for methods *diagbroyden*, *linearmixing* and *excitingmixing* may be useful for specific problems, but whether they will work may depend strongly on the problem.

New in version 0.11.0.

References

[R185], [R186], [R187]

Examples

The following functions define a system of nonlinear equations and its jacobian.

```
>>> def fun(x):
...     return [x[0]  + 0.5 * (x[0] - x[1])**3 - 1.0,
...             0.5 * (x[1] - x[0])**3 + x[1]]
```

```
>>> def jac(x):
...     return np.array([[1 + 1.5 * (x[0] - x[1])**2,
...                       -1.5 * (x[0] - x[1])**2],
...                      [-1.5 * (x[1] - x[0])**2,
...                       1 + 1.5 * (x[1] - x[0])**2]])
```

A solution can be obtained as follows.

```
>>> from scipy import optimize
>>> sol = optimize.root(fun, [0, 0], jac=jac, method='hybr')
>>> sol.x
array([ 0.8411639,  0.1588361])
```

`scipy.optimize.`**`fsolve`**(*func*, *x0*, *args=()*, *fprime=None*, *full_output=0*, *col_deriv=0*, *xtol=1.49012e-08*, *maxfev=0*, *band=None*, *epsfcn=None*, *factor=100*, *diag=None*)

Find the roots of a function.

Return the roots of the (non-linear) equations defined by `func(x) = 0` given a starting estimate.

Parameters

func : callable `f(x, *args)`
A function that takes at least one (possibly vector) argument.

x0 : ndarray
The starting estimate for the roots of `func(x) = 0`.

args : tuple, optional
Any extra arguments to *func*.

fprime : callable(x), optional
A function to compute the Jacobian of *func* with derivatives across the rows. By default, the Jacobian will be estimated.

full_output : bool, optional
If True, return optional outputs.

col_deriv : bool, optional
Specify whether the Jacobian function computes derivatives down the columns (faster, because there is no transpose operation).

xtol : float, optional

The calculation will terminate if the relative error between two consecutive iterates is at most *xtol*.

maxfev : int, optional
The maximum number of calls to the function. If zero, then `100*(N+1)` is the maximum where N is the number of elements in *x0*.

band : tuple, optional
If set to a two-sequence containing the number of sub- and super-diagonals within the band of the Jacobi matrix, the Jacobi matrix is considered banded (only for `fprime=None`).

epsfcn : float, optional
A suitable step length for the forward-difference approximation of the Jacobian (for `fprime=None`). If *epsfcn* is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

factor : float, optional
A parameter determining the initial step bound (`factor * || diag * x||`). Should be in the interval `(0.1, 100)`.

diag : sequence, optional
N positive entries that serve as a scale factors for the variables.

Returns

x : ndarray
The solution (or the result of the last iteration for an unsuccessful call).

infodict : dict
A dictionary of optional outputs with the keys:

nfev	number of function calls
njev	number of Jacobian calls
fvec	function evaluated at the output
fjac	the orthogonal matrix, q, produced by the QR factorization of the final approximate Jacobian matrix, stored column wise
r	upper triangular matrix produced by QR factorization of the same matrix
qtf	the vector `(transpose(q) * fvec)`

ier : int
An integer flag. Set to 1 if a solution was found, otherwise refer to *mesg* for more information.

mesg : str
If no solution is found, *mesg* details the cause of failure.

See also:

`root` Interface to root finding algorithms for multivariate

`functions.`

Notes

`fsolve` is a wrapper around MINPACK's hybrd and hybrj algorithms.

`scipy.optimize.`**`broyden1`**(*F*, *xin*, *iter=None*, *alpha=None*, *reduction_method='restart'*, *max_rank=None*, *verbose=False*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *tol_norm=None*, *line_search='armijo'*, *callback=None*, ***kw*)

Find a root of a function, using Broyden's first Jacobian approximation.

This method is also known as "Broyden's good method".

Parameters

F : function(x) -> f
Function whose root to find; should take and return an array-like object.

x0 : array_like
Initial guess for the solution

alpha : float, optional
Initial guess for the Jacobian is `(-1/alpha)`.

reduction_method : str or tuple, optional
Method used in ensuring that the rank of the Broyden matrix stays low. Can either be a string giving the name of the method, or a tuple of the form `(method, param1, param2, ...)` that gives the name of the method and values for additional parameters.
Methods available:

- `restart`: drop all matrix columns. Has no extra parameters.
- `simple`: drop oldest matrix column. Has no extra parameters.
- `svd`: keep only the most significant SVD components. Takes an extra parameter, `to_retain`, which determines the number of SVD components to retain when rank reduction is done. Default is `max_rank - 2`.

max_rank : int, optional
Maximum rank for the Broyden matrix. Default is infinity (ie., no rank reduction).

iter : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional
Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns

sol : ndarray
An array (of similar array type as *x0*) containing the final solution.

Raises

NoConvergence
When a solution was not found.

Notes

This algorithm implements the inverse Jacobian Quasi-Newton update

$$H_+ = H + (dx - H df) dx^\dagger H / (dx^\dagger H df)$$

which corresponds to Broyden's first Jacobian update

$$J_+ = J + (df - J dx) dx^\dagger / dx^\dagger dx$$

References

[R162]

`scipy.optimize.broyden2(F, xin, iter=None, alpha=None, reduction_method='restart', max_rank=None, verbose=False, maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, tol_norm=None, line_search='armijo', callback=None, **kw)`

Find a root of a function, using Broyden's second Jacobian approximation.

This method is also known as "Broyden's bad method".

Parameters

F : function(x) -> f
: Function whose root to find; should take and return an array-like object.

x0 : array_like
: Initial guess for the solution

alpha : float, optional
: Initial guess for the Jacobian is `(-1/alpha)`.

reduction_method : str or tuple, optional
: Method used in ensuring that the rank of the Broyden matrix stays low. Can either be a string giving the name of the method, or a tuple of the form `(method, param1, param2, ...)` that gives the name of the method and values for additional parameters.
: Methods available:
  - `restart`: drop all matrix columns. Has no extra parameters.
  - `simple`: drop oldest matrix column. Has no extra parameters.
  - `svd`: keep only the most significant SVD components. Takes an extra parameter, `to_retain`, which determines the number of SVD components to retain when rank reduction is done. Default is `max_rank - 2`.

max_rank : int, optional
: Maximum rank for the Broyden matrix. Default is infinity (ie., no rank reduction).

iter : int, optional
: Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
: Print status to stdout on every iteration.

maxiter : int, optional
: Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
: Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
: Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
: Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
: Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
: Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
: Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional

Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns **sol** : ndarray

An array (of similar array type as *x0*) containing the final solution.

Raises **NoConvergence**

When a solution was not found.

Notes

This algorithm implements the inverse Jacobian Quasi-Newton update

$$H_+ = H + (dx - H df) df^\dagger / (df^\dagger df)$$

corresponding to Broyden's second method.

References

[R163]

The `root` function supports the following methods:

root(method='hybr')

`scipy.optimize.`**`root`**(*fun*, *x0*, *args=()*, *method='hybr'*, *jac=None*, *tol=None*, *callback=None*, *options={'col_deriv': 0, 'diag': None, 'factor': 100, 'eps': None, 'band': None, 'func': None, 'maxfev': 0, 'xtol': 1.49012e-08}*)

Find the roots of a multivariate function using MINPACK's hybrd and hybrj routines (modified Powell method).

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options **col_deriv** : bool

Specify whether the Jacobian function computes derivatives down the columns (faster, because there is no transpose operation).

xtol : float

The calculation will terminate if the relative error between two consecutive iterates is at most *xtol*.

maxfev : int

The maximum number of calls to the function. If zero, then `100*(N+1)` is the maximum where N is the number of elements in *x0*.

band : tuple

If set to a two-sequence containing the number of sub- and super-diagonals within the band of the Jacobi matrix, the Jacobi matrix is considered banded (only for `fprime=None`).

eps : float

A suitable step length for the forward-difference approximation of the Jacobian (for `fprime=None`). If *eps* is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

factor : float

A parameter determining the initial step bound (`factor * || diag * x||`). Should be in the interval `(0.1, 100)`.

diag : sequence

N positive entries that serve as a scale factors for the variables.

root(method='lm')

`scipy.optimize.`**`root`**(*fun*, *x0*, *args=()*, *method='lm'*, *jac=None*, *tol=None*, *callback=None*, *options={'col_deriv': 0, 'diag': None, 'factor': 100, 'gtol': 0.0, 'eps': 0.0, 'func': None, 'maxiter': 0, 'xtol': 1.49012e-08, 'ftol': 1.49012e-08}*)

Solve for least squares with Levenberg-Marquardt

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

col_deriv : bool
non-zero to specify that the Jacobian function computes derivatives down the columns (faster, because there is no transpose operation).

ftol : float
Relative error desired in the sum of squares.

xtol : float
Relative error desired in the approximate solution.

gtol : float
Orthogonality desired between the function vector and the columns of the Jacobian.

maxiter : int
The maximum number of calls to the function. If zero, then 100*(N+1) is the maximum where N is the number of elements in x0.

epsfcn : float
A suitable step length for the forward-difference approximation of the Jacobian (for Dfun=None). If epsfcn is less than the machine precision, it is assumed that the relative errors in the functions are of the order of the machine precision.

factor : float
A parameter determining the initial step bound (`factor * || diag * x||`). Should be in interval `(0.1, 100)`.

diag : sequence
N positive entries that serve as a scale factors for the variables.

root(method='broyden1')

`scipy.optimize.`**`root`**(*fun*, *x0*, *args=()*, *method='broyden1'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional

Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] Initial guess for the Jacobian is (-1/alpha).

reduction_method [str or tuple, optional] Method used in ensuring that the rank of the Broyden matrix stays low. Can either be a string giving the name of the method, or a tuple of the form `(method, param1, param2, ...)` that gives the name of the method and values for additional parameters.

Methods available:

- ***`restart`: drop all matrix columns. Has no*** extra parameters.
- ***`simple`: drop oldest matrix column. Has no*** extra parameters.
- ***`svd`: keep only the most significant SVD*** components.

Extra parameters:

- ***`to_retain`: number of SVD compon***

retain when rank reduction is done. Default is `max_rank - 2`.

max_rank [int, optional] Maximum rank for the Broyden matrix. Default is infinity (ie., no rank reduction).

root(method='broyden2')

`scipy.optimize.`**`root`** (*fun*, *x0*, *args=()*, *method='broyden2'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] Initial guess for the Jacobian is (-1/alpha).

reduction_method
[str or tuple, optional] Method used in ensuring that the rank of the Broyden matrix stays low. Can either be a string giving the name of the method, or a tuple of the form `(method, param1, param2, ...)` that gives the name of the method and values for additional parameters.

Methods available:

- **`restart`**: *drop all matrix columns. Has no*
  extra parameters.
- **`simple`**: *drop oldest matrix column. Has no*
  extra parameters.
- **`svd`**: *keep only the most significant SVD*
  components.

Extra parameters:

- **`to_retain`**: *number of SVD compo*
  retain
  when
  rank

reduction is done. Default is max_rank - 2.

max_rank [int, optional] Maximum rank for the Broyden matrix. Default is infinity (ie., no rank reduction).

root(method='anderson')

scipy.optimize.**root**(*fun*, *x0*, *args=()*, *method='anderson'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.root*

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] Initial guess for the Jacobian is (-1/alpha).

M [float, optional] Number of previous vectors to retain. Defaults to 5.

w0 [float, optional] Regularization parameter for numerical stability. Compared to unity, good values of the order of 0.01.

root(method='linearmixing')

`scipy.optimize.root` (*fun*, *x0*, *args=()*, *method='linearmixing'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.root*

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, `NoConvergence` is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] initial guess for the jacobian is (-1/alpha).

root(method='diagbroyden')

`scipy.optimize.root` (*fun*, *x0*, *args=()*, *method='diagbroyden'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see *scipy.optimize.root*

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional

Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] initial guess for the jacobian is (-1/alpha).

root(method='excitingmixing')

`scipy.optimize.root`(*fun*, *x0*, *args=()*, *method='excitingmixing'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

alpha [float, optional] Initial Jacobian approximation is (-1/alpha).

alphamax [float, optional] The entries of the diagonal Jacobian are kept in the range `[alpha, alphamax]`.

root(method='krylov')

`scipy.optimize.root` (*fun*, *x0*, *args=()*, *method='krylov'*, *tol=None*, *callback=None*, *options={}*)

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

nit : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

disp : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

ftol : float, optional
Relative tolerance for the residual. If omitted, not used.

fatol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

xtol : float, optional
Relative minimum step size. If omitted, not used.

xatol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

jac_options : dict, optional
Options for the respective Jacobian approximation.

- ***rdiff*** [float, optional] Relative step size to use in numerical differentiation.
- ***method*** [{'lgmres', 'gmres', 'bicgstab', 'cgs', 'minres'} or function] Krylov method to use to approximate the Jacobian. Can be a string, or a function implementing the same interface as the iterative solvers in `scipy.sparse.linalg`. The default is `scipy.sparse.linalg.lgmres`.
- ***inner_M*** [LinearOperator or InverseJacobian] Preconditioner for the inner Krylov iteration. Note that you can use also inverse Jacobians as (adaptive) preconditioners. For example,

```
>>> jac = BroydenFirst()
>>> kjac = KrylovJacobian(inner_M=jac.
↪inverse).
```

  If the preconditioner has a method named 'update', it will be called as `update(x, f)` after each nonlinear step, with `x` giving the current point, and `f` the current function value.
- ***inner_tol, inner_maxiter, ...*** Parameters to pass on to the "inner" Krylov solver. See `scipy.sparse.linalg.gmres` for details.
- ***outer_k*** [int, optional] Size of the subspace kept across LGMRES nonlinear iterations.

See `scipy.sparse.linalg.lgmres` for details.

root(method='df-sane')

`scipy.optimize.`**`root`**(*fun*, *x0*, *args=()*, *method='df-sane'*, *tol=None*, *callback=None*, *options={'disp': False*, *'fnorm': None*, *'sigma_0': 1.0*, *'eta_strategy': None*, *'sigma_eps': 1e-10*, *'M': 10*, *'line_search': 'cruz'*, *'fatol': 1e-300*, *'func': None*, *'maxfev': 1000*, *'ftol': 1e-08}*)

Solve nonlinear equation with the DF-SANE method

See also:

For documentation for the rest of the parameters, see `scipy.optimize.root`

Options

ftol : float, optional
Relative norm tolerance.

fatol : float, optional
Absolute norm tolerance. Algorithm terminates when `||func(x)|| < fatol + ftol ||func(x_0)||`.

fnorm : callable, optional
Norm to use in the convergence check. If None, 2-norm is used.

maxfev : int, optional
Maximum number of function evaluations.

disp : bool, optional
Whether to print convergence process to stdout.

eta_strategy : callable, optional
Choice of the `eta_k` parameter, which gives slack for growth of `||F||**2`. Called as `eta_k = eta_strategy(k, x, F)` with *k* the iteration number, *x* the current iterate and *F* the current residual. Should satisfy `eta_k > 0` and `sum(eta, k=0..inf) < inf`. Default: `||F||**2 / (1 + k)**2`.

sigma_eps : float, optional
The spectral coefficient is constrained to `sigma_eps < sigma < 1/sigma_eps`. Default: 1e-10

sigma_0 : float, optional
Initial spectral coefficient. Default: 1.0

M : int, optional
Number of iterates to include in the nonmonotonic line search. Default: 10

line_search : {'cruz', 'cheng'}
Type of line search to employ. 'cruz' is the original one defined in [Martinez & Raydan. Math. Comp. 75, 1429 (2006)], 'cheng' is a modified search defined in [Cheng & Li. IMA J. Numer. Anal. 29, 814 (2009)]. Default: 'cruz'

References

[R813], *[R814]*, *[R815]*

Large-scale nonlinear solvers:

`newton_krylov`(F, xin[, iter, rdiff, method, ...])	Find a root of a function, using Krylov approximation for inverse Jacobian.
`anderson`(F, xin[, iter, alpha, w0, M, ...])	Find a root of a function, using (extended) Anderson mixing.

`scipy.optimize.`**`newton_krylov`**(*F*, *xin*, *iter=None*, *rdiff=None*, *method='lgmres'*, *inner_maxiter=20*, *inner_M=None*, *outer_k=10*, *verbose=False*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *tol_norm=None*, *line_search='armijo'*, *callback=None*, ***kw*)

Find a root of a function, using Krylov approximation for inverse Jacobian.

This method is suitable for solving large-scale problems.

Parameters

F : function(x) -> f
: Function whose root to find; should take and return an array-like object.

x0 : array_like
: Initial guess for the solution

rdiff : float, optional
: Relative step size to use in numerical differentiation.

method : {'lgmres', 'gmres', 'bicgstab', 'cgs', 'minres'} or function
: Krylov method to use to approximate the Jacobian. Can be a string, or a function implementing the same interface as the iterative solvers in *`scipy.sparse.linalg`*.
: The default is *`scipy.sparse.linalg.lgmres`*.

inner_M : LinearOperator or InverseJacobian
: Preconditioner for the inner Krylov iteration. Note that you can use also inverse Jacobians as (adaptive) preconditioners. For example,

```
>>> from scipy.optimize.nonlin import BroydenFirst, KrylovJacobian
>>> from scipy.optimize.nonlin import InverseJacobian
>>> jac = BroydenFirst()
>>> kjac = KrylovJacobian(inner_M=InverseJacobian(jac))
```

: If the preconditioner has a method named 'update', it will be called as `update(x, f)` after each nonlinear step, with `x` giving the current point, and `f` the current function value.

inner_tol, inner_maxiter, ...
: Parameters to pass on to the "inner" Krylov solver. See *`scipy.sparse.linalg.gmres`* for details.

outer_k : int, optional
: Size of the subspace kept across LGMRES nonlinear iterations. See *`scipy.sparse.linalg.lgmres`* for details.

iter : int, optional
: Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
: Print status to stdout on every iteration.

maxiter : int, optional
: Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
: Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
: Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
: Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional
Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns **sol** : ndarray
An array (of similar array type as *x0*) containing the final solution.

Raises **NoConvergence**
When a solution was not found.

See also:

scipy.sparse.linalg.gmres, *scipy.sparse.linalg.lgmres*

Notes

This function implements a Newton-Krylov solver. The basic idea is to compute the inverse of the Jacobian with an iterative Krylov method. These methods require only evaluating the Jacobian-vector products, which are conveniently approximated by a finite difference:

$$Jv \approx (f(x + \omega * v/|v|) - f(x))/\omega$$

Due to the use of iterative matrix inverses, these methods can deal with large nonlinear problems.

Scipy's *scipy.sparse.linalg* module offers a selection of Krylov solvers to choose from. The default here is *lgmres*, which is a variant of restarted GMRES iteration that reuses some of the information obtained in the previous Newton steps to invert Jacobians in subsequent steps.

For a review on Newton-Krylov methods, see for example *[R183]*, and for the LGMRES sparse inverse method, see *[R184]*.

References

[R183], *[R184]*

`scipy.optimize.`**`anderson`**(*F, xin, iter=None, alpha=None, w0=0.01, M=5, verbose=False, maxiter=None, f_tol=None, f_rtol=None, x_tol=None, x_rtol=None, tol_norm=None, line_search='armijo', callback=None, **kw*)

Find a root of a function, using (extended) Anderson mixing.

The Jacobian is formed by for a 'best' solution in the space spanned by last *M* vectors. As a result, only a MxM matrix inversions and MxN multiplications are required. *[Ey]*

Parameters **F** : function(x) -> f
Function whose root to find; should take and return an array-like object.

x0 : array_like
Initial guess for the solution

alpha : float, optional
Initial guess for the Jacobian is (-1/alpha).

M : float, optional
Number of previous vectors to retain. Defaults to 5.

w0 : float, optional
Regularization parameter for numerical stability. Compared to unity, good values of the order of 0.01.

iter : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional
Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns **sol** : ndarray
An array (of similar array type as *x0*) containing the final solution.

Raises **NoConvergence**
When a solution was not found.

References

[Ey]

Simple iterations:

`excitingmixing`(F, xin[, iter, alpha, ...])	Find a root of a function, using a tuned diagonal Jacobian approximation.
`linearmixing`(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using a scalar Jacobian approximation.
`diagbroyden`(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using diagonal Broyden Jacobian approximation.

`scipy.optimize.`**`excitingmixing`**(*F*, *xin*, *iter=None*, *alpha=None*, *alphamax=1.0*, *verbose=False*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *tol_norm=None*, *line_search='armijo'*, *callback=None*, ***kw*)

Find a root of a function, using a tuned diagonal Jacobian approximation.

The Jacobian matrix is diagonal and is tuned on each iteration.

> **Warning:** This algorithm may be useful for specific problems, but whether it will work may depend strongly on the problem.

Parameters

F : function(x) -> f
: Function whose root to find; should take and return an array-like object.

x0 : array_like
: Initial guess for the solution

alpha : float, optional
: Initial Jacobian approximation is (-1/alpha).

alphamax : float, optional
: The entries of the diagonal Jacobian are kept in the range `[alpha, alphamax]`.

iter : int, optional
: Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
: Print status to stdout on every iteration.

maxiter : int, optional
: Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
: Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
: Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
: Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
: Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
: Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
: Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional
: Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns

sol : ndarray
: An array (of similar array type as *x0*) containing the final solution.

Raises

NoConvergence
: When a solution was not found.

`scipy.optimize.`**`linearmixing`**(*F*, *xin*, *iter=None*, *alpha=None*, *verbose=False*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *tol_norm=None*, *line_search='armijo'*, *callback=None*, ***kw*)

Find a root of a function, using a scalar Jacobian approximation.

> **Warning:** This algorithm may be useful for specific problems, but whether it will work may depend strongly on the problem.

Parameters **F** : function(x) -> f

Function whose root to find; should take and return an array-like object.

x0 : array_like

Initial guess for the solution

alpha : float, optional

The Jacobian approximation is (-1/alpha).

iter : int, optional

Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional

Print status to stdout on every iteration.

maxiter : int, optional

Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional

Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional

Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional

Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional

Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional

Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional

Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional

Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns **sol** : ndarray

An array (of similar array type as *x0*) containing the final solution.

Raises **NoConvergence**

When a solution was not found.

`scipy.optimize.`**`diagbroyden`**(*F*, *xin*, *iter=None*, *alpha=None*, *verbose=False*, *maxiter=None*, *f_tol=None*, *f_rtol=None*, *x_tol=None*, *x_rtol=None*, *tol_norm=None*, *line_search='armijo'*, *callback=None*, ***kw*)

Find a root of a function, using diagonal Broyden Jacobian approximation.

The Jacobian approximation is derived from previous iterations, by retaining only the diagonal of Broyden matrices.

Warning: This algorithm may be useful for specific problems, but whether it will work may depend strongly on the problem.

Parameters **F** : function(x) -> f

Function whose root to find; should take and return an array-like object.

x0 : array_like

Initial guess for the solution

alpha : float, optional
Initial guess for the Jacobian is (-1/alpha).

iter : int, optional
Number of iterations to make. If omitted (default), make as many as required to meet tolerances.

verbose : bool, optional
Print status to stdout on every iteration.

maxiter : int, optional
Maximum number of iterations to make. If more are needed to meet convergence, *NoConvergence* is raised.

f_tol : float, optional
Absolute tolerance (in max-norm) for the residual. If omitted, default is 6e-6.

f_rtol : float, optional
Relative tolerance for the residual. If omitted, not used.

x_tol : float, optional
Absolute minimum step size, as determined from the Jacobian approximation. If the step size is smaller than this, optimization is terminated as successful. If omitted, not used.

x_rtol : float, optional
Relative minimum step size. If omitted, not used.

tol_norm : function(vector) -> scalar, optional
Norm to use in convergence check. Default is the maximum norm.

line_search : {None, 'armijo' (default), 'wolfe'}, optional
Which type of a line search to use to determine the step size in the direction given by the Jacobian approximation. Defaults to 'armijo'.

callback : function, optional
Optional callback function. It is called on every iteration as `callback(x, f)` where *x* is the current solution and *f* the corresponding residual.

Returns **sol** : ndarray
An array (of similar array type as *x0*) containing the final solution.

Raises **NoConvergence**
When a solution was not found.

Additional information on the nonlinear solvers

5.18.4 Linear Programming

Simplex Algorithm:

linprog(c[, A_ub, b_ub, A_eq, b_eq, bounds, ...])	Minimize a linear objective function subject to linear equality and inequality constraints.
linprog_verbose_callback(xk, **kwargs)	A sample callback function demonstrating the linprog callback interface.

`scipy.optimize.`**`linprog`**(*c*, *A_ub=None*, *b_ub=None*, *A_eq=None*, *b_eq=None*, *bounds=None*, *method='simplex'*, *callback=None*, *options=None*)

Minimize a linear objective function subject to linear equality and inequality constraints.

Linear Programming is intended to solve the following problem form:

Minimize: c^T * x
Subject to: A_ub * x <= b_ub
A_eq * x == b_eq

Parameters

c : array_like
Coefficients of the linear objective function to be minimized.

A_ub : array_like, optional
2-D array which, when matrix-multiplied by x, gives the values of the upper-bound inequality constraints at x.

b_ub : array_like, optional
1-D array of values representing the upper-bound of each inequality constraint (row) in A_ub.

A_eq : array_like, optional
2-D array which, when matrix-multiplied by x, gives the values of the equality constraints at x.

b_eq : array_like, optional
1-D array of values representing the RHS of each equality constraint (row) in A_eq.

bounds : sequence, optional
`(min, max)` pairs for each element in `x`, defining the bounds on that parameter. Use None for one of `min` or `max` when there is no bound in that direction. By default bounds are `(0, None)` (non-negative) If a sequence containing a single tuple is provided, then `min` and `max` will be applied to all variables in the problem.

method : str, optional
Type of solver. At this time only 'simplex' is supported *(see here)*.

callback : callable, optional
If a callback function is provide, it will be called within each iteration of the simplex algorithm. The callback must have the signature *callback(xk, **kwargs)* where xk is the current solution vector and kwargs is a dictionary containing the following:

```
"tableau" : The current Simplex algorithm tableau
"nit" : The current iteration.
"pivot" : The pivot (row, column) used for the next␣
↪iteration.
"phase" : Whether the algorithm is in Phase 1 or Phase␣
↪2.
"basis" : The indices of the columns of the basic␣
↪variables.
```

options : dict, optional
A dictionary of solver options. All methods accept the following generic options:

maxiter [int] Maximum number of iterations to perform.

disp [bool] Set to True to print convergence messages.

For method-specific options, see *show_options('linprog')*.

Returns

A `scipy.optimize.OptimizeResult` consisting of the following fields:

x [ndarray] The independent variable vector which optimizes the linear programming problem.

fun [float] Value of the objective function.

slack [ndarray] The values of the slack variables. Each slack variable corresponds to an inequality constraint. If the slack is zero, then the corresponding constraint is active.

success [bool] Returns True if the algorithm succeeded in finding an optimal solution.

status [int] An integer representing the exit status of the optimization:

```
0 : Optimization terminated successfully
1 : Iteration limit reached
2 : Problem appears to be infeasible
3 : Problem appears to be unbounded
```

nit [int] The number of iterations performed.

message [str] A string descriptor of the exit status of the optimization.

See also:

show_options Additional options accepted by the solvers

Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is *Simplex*.

Method *Simplex* uses the Simplex algorithm (as it relates to Linear Programming, NOT the Nelder-Mead Simplex) *[R171]*, *[R172]*. This algorithm should be reasonably reliable and fast.

New in version 0.15.0.

References

[R171], *[R172]*, *[R173]*

Examples

Consider the following problem:

Minimize: f = -1*x[0] + 4*x[1]
Subject to: -3*x[0] + 1*x[1] <= 6

1*x[0] + 2*x[1] <= 4
x[1] >= -3
where: -inf <= x[0] <= inf

This problem deviates from the standard linear programming problem. In standard form, linear programming problems assume the variables x are non-negative. Since the variables don't have standard bounds where 0 <= x <= inf, the bounds of the variables must be explicitly set.

There are two upper-bound constraints, which can be expressed as

dot(A_ub, x) <= b_ub

The input for this problem is as follows:

```
>>> c = [-1, 4]
>>> A = [[-3, 1], [1, 2]]
>>> b = [6, 4]
>>> x0_bounds = (None, None)
>>> x1_bounds = (-3, None)
>>> from scipy.optimize import linprog
>>> res = linprog(c, A_ub=A, b_ub=b, bounds=(x0_bounds, x1_bounds),
...               options={"disp": True})
Optimization terminated successfully.
     Current function value: -22.000000
     Iterations: 1
>>> print(res)
     fun: -22.0
 message: 'Optimization terminated successfully.'
```

```
    nit: 1
  slack: array([ 39.,   0.])
 status: 0
success: True
      x: array([ 10.,  -3.])
```

Note the actual objective value is 11.428571. In this case we minimized the negative of the objective function.

`scipy.optimize.linprog_verbose_callback(xk, **kwargs)`

A sample callback function demonstrating the linprog callback interface. This callback produces detailed output to sys.stdout before each iteration and after the final iteration of the simplex algorithm.

Parameters

xk : array_like

The current solution vector.

****kwargs** : dict

A dictionary containing the following parameters:

tableau [array_like] The current tableau of the simplex algorithm. Its structure is defined in _solve_simplex.

phase [int] The current Phase of the simplex algorithm (1 or 2)

nit [int] The current iteration number.

pivot [tuple(int, int)] The index of the tableau selected as the next pivot, or nan if no pivot exists

basis [array(int)] A list of the current basic variables. Each element contains the name of a basic variable and its value.

complete [bool] True if the simplex algorithm has completed (and this is the final call to callback), otherwise False.

The `linprog` function supports the following methods:

linprog(method='Simplex')

`scipy.optimize.linprog(c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='simplex', callback=None, options={'disp': False, 'bland': False, 'tol': 1e-12, 'maxiter': 1000})`

Solve the following linear programming problem via a two-phase simplex algorithm.

minimize: c^T * x

subject to: A_ub * x <= b_ub

A_eq * x == b_eq

Parameters

c : array_like

Coefficients of the linear objective function to be minimized.

A_ub : array_like

2-D array which, when matrix-multiplied by x, gives the values of the upper-bound inequality constraints at x.

b_ub : array_like

1-D array of values representing the upper-bound of each inequality constraint (row) in A_ub.

A_eq : array_like

2-D array which, when matrix-multiplied by x, gives the values of the equality constraints at x.

b_eq : array_like

1-D array of values representing the RHS of each equality constraint (row) in A_eq.

bounds : array_like

The bounds for each independent variable in the solution, which can take one of three forms:: None : The default bounds, all variables are non-negative. (lb, ub) : If a 2-element sequence is provided, the same

lower bound (lb) and upper bound (ub) will be applied to all variables.

[(lb_0, ub_0), (lb_1, ub_1), ...]
[If an n x 2 sequence is provided,] each variable x_i will be bounded by lb[i] and ub[i].

Infinite bounds are specified using -np.inf (negative) or np.inf (positive).

callback : callable
If a callback function is provide, it will be called within each iteration of the simplex algorithm. The callback must have the signature *callback(xk, **kwargs)* where xk is the current solution vector and kwargs is a dictionary containing the following:: "tableau" : The current Simplex algorithm tableau "nit" : The current iteration. "pivot" : The pivot (row, column) used for the next iteration. "phase" : Whether the algorithm is in Phase 1 or Phase 2. "bv" : A structured array containing a string representation of each

basic variable and its current value.

Returns A scipy.optimize.OptimizeResult consisting of the following fields:

```
x : ndarray
    The independent variable vector which optimizes the linear
    programming problem.
fun : float
    Value of the objective function.
slack : ndarray
    The values of the slack variables.  Each slack variable␣
↪corresponds
    to an inequality constraint.  If the slack is zero, then the
    corresponding constraint is active.
success : bool
    Returns True if the algorithm succeeded in finding an optimal
    solution.
status : int
    An integer representing the exit status of the optimization::
     0 : Optimization terminated successfully
     1 : Iteration limit reached
     2 : Problem appears to be infeasible
     3 : Problem appears to be unbounded
nit : int
    The number of iterations performed.
message : str
    A string descriptor of the exit status of the optimization.
```

See also:

For documentation for the rest of the parameters, see `scipy.optimize.linprog`

Options **maxiter** : int
The maximum number of iterations to perform.

disp : bool
If True, print exit status message to sys.stdout

tol : float
The tolerance which determines when a solution is "close enough" to zero in Phase 1 to be considered a basic feasible solution or close enough to positive to to serve as an optimal solution.

bland : bool

If True, use Bland's anti-cycling rule [3] to choose pivots to prevent cycling. If False, choose pivots which should lead to a converged solution more quickly. The latter method is subject to cycling (non-convergence) in rare instances.

References

[R806], [R807], [R808]

Examples

Consider the following problem:

Minimize: f = -1*x[0] + 4*x[1]
Subject to: -3*x[0] + 1*x[1] <= 6

1*x[0] + 2*x[1] <= 4
x[1] >= -3
where: -inf <= x[0] <= inf

This problem deviates from the standard linear programming problem. In standard form, linear programming problems assume the variables x are non-negative. Since the variables don't have standard bounds where 0 <= x <= inf, the bounds of the variables must be explicitly set.

There are two upper-bound constraints, which can be expressed as

dot(A_ub, x) <= b_ub

The input for this problem is as follows:

```
>>> from scipy.optimize import linprog
>>> c = [-1, 4]
>>> A = [[-3, 1], [1, 2]]
>>> b = [6, 4]
>>> x0_bnds = (None, None)
>>> x1_bnds = (-3, None)
>>> res = linprog(c, A, b, bounds=(x0_bnds, x1_bnds))
>>> print(res)
     fun: -22.0
 message: 'Optimization terminated successfully.'
     nit: 1
   slack: array([ 39.,   0.])
  status: 0
 success: True
       x: array([ 10.,  -3.])
```

Assignment problems:

linear_sum_assignment(cost_matrix)	Solve the linear sum assignment problem.

scipy.optimize.**linear_sum_assignment**(*cost_matrix*)
Solve the linear sum assignment problem.

The linear sum assignment problem is also known as minimum weight matching in bipartite graphs. A problem instance is described by a matrix C, where each C[i,j] is the cost of matching vertex i of the first partite set (a "worker") and vertex j of the second set (a "job"). The goal is to find a complete assignment of workers to jobs of minimal cost.

Formally, let X be a boolean matrix where $X[i,j] = 1$ iff row i is assigned to column j. Then the optimal

assignment has cost

$$\min \sum_i \sum_j C_{i,j} X_{i,j}$$

s.t. each row is assignment to at most one column, and each column to at most one row.

This function can also solve a generalization of the classic assignment problem where the cost matrix is rectangular. If it has more rows than columns, then not every row needs to be assigned to a column, and vice versa.

The method used is the Hungarian algorithm, also known as the Munkres or Kuhn-Munkres algorithm.

Parameters **cost_matrix** : array

The cost matrix of the bipartite graph.

Returns **row_ind, col_ind** : array

An array of row indices and one of corresponding column indices giving the optimal assignment. The cost of the assignment can be computed as `cost_matrix[row_ind, col_ind].sum()`. The row indices will be sorted; in the case of a square cost matrix they will be equal to `numpy.arange(cost_matrix.shape[0])`.

Notes

New in version 0.17.0.

References

1.http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html
2.Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83-97, 1955.
3.Harold W. Kuhn. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3: 253-258, 1956.
4.Munkres, J. Algorithms for the Assignment and Transportation Problems. *J. SIAM*, 5(1):32-38, March, 1957.
5.https://en.wikipedia.org/wiki/Hungarian_algorithm

Examples

```
>>> cost = np.array([[4, 1, 3], [2, 0, 5], [3, 2, 2]])
>>> from scipy.optimize import linear_sum_assignment
>>> row_ind, col_ind = linear_sum_assignment(cost)
>>> col_ind
array([1, 0, 2])
>>> cost[row_ind, col_ind].sum()
5
```

5.18.5 Utilities

approx_fprime(xk, f, epsilon, *args)	Finite-difference approximation of the gradient of a scalar function.
bracket(func[, xa, xb, args, grow_limit, ...])	Bracket the minimum of the function.
check_grad(func, grad, x0, *args, **kwargs)	Check the correctness of a gradient function by comparing it against a (forward) finite-difference approximation of the gradient.

Continued on next page

Table 5.118 – continued from previous page

line_search(f, myfprime, xk, pk[, gfk, ...])	Find alpha that satisfies strong Wolfe conditions.
show_options([solver, method, disp])	Show documentation for additional options of optimization solvers.
LbfgsInvHessProduct(sk, yk)	Linear operator for the L-BFGS approximate inverse Hessian.

scipy.optimize.**approx_fprime** (*xk, f, epsilon, *args*)

Finite-difference approximation of the gradient of a scalar function.

Parameters

xk : array_like
The coordinate vector at which to determine the gradient of *f*.

f : callable
The function of which to determine the gradient (partial derivatives). Should take *xk* as first argument, other arguments to *f* can be supplied in `*args`. Should return a scalar, the value of the function at *xk*.

epsilon : array_like
Increment to *xk* to use for determining the function gradient. If a scalar, uses the same finite difference delta for all partial derivatives. If an array, should contain one value per element of *xk*.

***args** : args, optional
Any other arguments that are to be passed to *f*.

Returns

grad : ndarray
The partial derivatives of *f* to *xk*.

See also:

check_grad Check correctness of gradient function against approx_fprime.

Notes

The function gradient is determined by the forward finite difference formula:

```
         f(xk[i] + epsilon[i]) - f(xk[i])
f'[i] = ---------------------------------
                    epsilon[i]
```

The main use of *approx_fprime* is in scalar function optimizers like *fmin_bfgs*, to determine numerically the Jacobian of a function.

Examples

```
>>> from scipy import optimize
>>> def func(x, c0, c1):
...     "Coordinate vector `x` should be an array of size two."
...     return c0 * x[0]**2 + c1*x[1]**2
```

```
>>> x = np.ones(2)
>>> c0, c1 = (1, 200)
>>> eps = np.sqrt(np.finfo(float).eps)
>>> optimize.approx_fprime(x, func, [eps, np.sqrt(200) * eps], c0, c1)
array([   2.        ,  400.00004198])
```

scipy.optimize.**bracket** (*func, xa=0.0, xb=1.0, args=(), grow_limit=110.0, maxiter=1000*)

Bracket the minimum of the function.

Given a function and distinct initial points, search in the downhill direction (as defined by the initial points) and return new points xa, xb, xc that bracket the minimum of the function f(xa) > f(xb) < f(xc). It doesn't always mean that obtained solution will satisfy xa<=x<=xb

Parameters

func : callable f(x,*args)
: Objective function to minimize.

xa, xb : float, optional
: Bracketing interval. Defaults *xa* to 0.0, and *xb* to 1.0.

args : tuple, optional
: Additional arguments (if present), passed to *func*.

grow_limit : float, optional
: Maximum grow limit. Defaults to 110.0

maxiter : int, optional
: Maximum number of iterations to perform. Defaults to 1000.

Returns

xa, xb, xc : float
: Bracket.

fa, fb, fc : float
: Objective function values in bracket.

funcalls : int
: Number of function evaluations made.

`scipy.optimize.check_grad(func, grad, x0, *args, **kwargs)`

Check the correctness of a gradient function by comparing it against a (forward) finite-difference approximation of the gradient.

Parameters

func : callable `func(x0, *args)`
: Function whose derivative is to be checked.

grad : callable `grad(x0, *args)`
: Gradient of *func*.

x0 : ndarray
: Points to check *grad* against forward difference approximation of grad using *func*.

args : *args, optional
: Extra arguments passed to *func* and *grad*.

epsilon : float, optional
: Step size used for the finite difference approximation. It defaults to `sqrt(numpy.finfo(float).eps)`, which is approximately 1.49e-08.

Returns

err : float
: The square root of the sum of squares (i.e. the 2-norm) of the difference between `grad(x0, *args)` and the finite difference approximation of *grad* using func at the points *x0*.

See also:

approx_fprime

Examples

```
>>> def func(x):
...     return x[0]**2 - 0.5 * x[1]**3
>>> def grad(x):
...     return [2 * x[0], -1.5 * x[1]**2]
>>> from scipy.optimize import check_grad
>>> check_grad(func, grad, [1.5, -1.5])
2.9802322387695312e-08
```

`scipy.optimize.`**`line_search`**(*f*, *myfprime*, *xk*, *pk*, *gfk=None*, *old_fval=None*, *old_old_fval=None*, *args=()*, *c1=0.0001*, *c2=0.9*, *amax=50*)

Find alpha that satisfies strong Wolfe conditions.

Parameters

f : callable f(x,*args)
: Objective function.

myfprime : callable f'(x,*args)
: Objective function gradient.

xk : ndarray
: Starting point.

pk : ndarray
: Search direction.

gfk : ndarray, optional
: Gradient value for x=xk (xk being the current parameter estimate). Will be recomputed if omitted.

old_fval : float, optional
: Function value for x=xk. Will be recomputed if omitted.

old_old_fval : float, optional
: Function value for the point preceding x=xk

args : tuple, optional
: Additional arguments passed to objective function.

c1 : float, optional
: Parameter for Armijo condition rule.

c2 : float, optional
: Parameter for curvature condition rule.

amax : float, optional
: Maximum step size

Returns

alpha : float or None
: Alpha for which `x_new = x0 + alpha * pk`, or None if the line search algorithm did not converge.

fc : int
: Number of function evaluations made.

gc : int
: Number of gradient evaluations made.

new_fval : float or None
: New function value `f(x_new)=f(x0+alpha*pk)`, or None if the line search algorithm did not converge.

old_fval : float
: Old function value `f(x0)`.

new_slope : float or None
: The local slope along the search direction at the new value `<myfprime(x_new), pk>`, or None if the line search algorithm did not converge.

Notes

Uses the line search algorithm to enforce strong Wolfe conditions. See Wright and Nocedal, 'Numerical Optimization', 1999, pg. 59-60.

For the zoom phase it uses an algorithm by [...].

`scipy.optimize.`**`show_options`**(*solver=None*, *method=None*, *disp=True*)

Show documentation for additional options of optimization solvers.

These are method-specific options that can be supplied through the `options` dict.

Parameters

solver : str

Type of optimization solver. One of 'minimize', 'minimize_scalar', 'root', or 'linprog'.

method : str, optional

If not given, shows all methods of the specified solver. Otherwise, show only the options for the specified method. Valid values corresponds to methods' names of respective solver (e.g. 'BFGS' for 'minimize').

disp : bool, optional

Whether to print the result rather than returning it.

Returns text

Either None (for disp=False) or the text string (disp=True)

Notes

The solver-specific methods are:

`scipy.optimize.minimize`

- *Nelder-Mead*
- *Powell*
- *CG*
- *BFGS*
- *Newton-CG*
- *L-BFGS-B*
- *TNC*
- *COBYLA*
- *SLSQP*
- *dogleg*
- *trust-ncg*

`scipy.optimize.root`

- *hybr*
- *lm*
- *broyden1*
- *broyden2*
- *anderson*
- *linearmixing*
- *diagbroyden*
- *excitingmixing*
- *krylov*
- *df-sane*

`scipy.optimize.minimize_scalar`

- *brent*
- *golden*
- *bounded*

`scipy.optimize.linprog`

- *simplex*

class `scipy.optimize.`**`LbfgsInvHessProduct`**(*sk*, *yk*)

Linear operator for the L-BFGS approximate inverse Hessian.

This operator computes the product of a vector with the approximate inverse of the Hessian of the objective function, using the L-BFGS limited memory approximation to the inverse Hessian, accumulated during the optimization.

Objects of this class implement the `scipy.sparse.linalg.LinearOperator` interface.

Parameters **sk** : array_like, shape=(n_corr, n)

Array of *n_corr* most recent updates to the solution vector. (See [1]).

yk : array_like, shape=(n_corr, n)

Array of *n_corr* most recent updates to the gradient. (See [1]).

References

[R157]

Attributes

H	Hermitian adjoint.
T	Transpose this linear operator.

`LbfgsInvHessProduct.`**`H`**

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** : LinearOperator
Hermitian adjoint of self.

`LbfgsInvHessProduct.`**`T`**

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

Methods

__call__(x)	
adjoint()	Hermitian adjoint.
dot(x)	Matrix-matrix or matrix-vector multiplication.
matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatvec(x)	Adjoint matrix-vector multiplication.
todense()	Return a dense array representation of this operator.
transpose()	Transpose this linear operator.

`LbfgsInvHessProduct.`**`__call__`**`(x)`

`LbfgsInvHessProduct.`**`adjoint`**`()`

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** : LinearOperator
Hermitian adjoint of self.

`LbfgsInvHessProduct.`**`dot`**`(x)`

Matrix-matrix or matrix-vector multiplication.

Parameters **x** : array_like
1-d or 2-d array, representing a vector or matrix.

Returns **Ax** : array

1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

LbfgsInvHessProduct.**matmat**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** : {matrix, ndarray}

An array with shape (N,K).

Returns **Y** : {matrix, ndarray}

A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

LbfgsInvHessProduct.**matvec**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}

An array with shape (N,) or (N,1).

Returns **y** : {matrix, ndarray}

A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

LbfgsInvHessProduct.**rmatvec**(*x*)

Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}

An array with shape (M,) or (M,1).

Returns **y** : {matrix, ndarray}

A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

LbfgsInvHessProduct.**todense**()

Return a dense array representation of this operator.

Returns **arr** : ndarray, shape=(n, n)

An array with the same shape and containing the same data represented by this *LinearOperator*.

LbfgsInvHessProduct.**transpose**()

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

5.19 Nonlinear solvers

This is a collection of general-purpose nonlinear multidimensional solvers. These solvers find *x* for which *F(x) = 0*. Both *x* and *F* can be multidimensional.

5.19.1 Routines

Large-scale nonlinear solvers:

newton_krylov(F, xin[, iter, rdiff, method, ...])	Find a root of a function, using Krylov approximation for inverse Jacobian.
anderson(F, xin[, iter, alpha, w0, M, ...])	Find a root of a function, using (extended) Anderson mixing.

General nonlinear solvers:

broyden1(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's first Jacobian approximation.
broyden2(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's second Jacobian approximation.

Simple iterations:

excitingmixing(F, xin[, iter, alpha, ...])	Find a root of a function, using a tuned diagonal Jacobian approximation.
linearmixing(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using a scalar Jacobian approximation.
diagbroyden(F, xin[, iter, alpha, verbose, ...])	Find a root of a function, using diagonal Broyden Jacobian approximation.

5.19.2 Examples

Small problem

```
>>> def F(x):
...     return np.cos(x) + x[::-1] - [1, 2, 3, 4]
>>> import scipy.optimize
>>> x = scipy.optimize.broyden1(F, [1,1,1,1], f_tol=1e-14)
>>> x
array([ 4.04674914,  3.91158389,  2.71791677,  1.61756251])
>>> np.cos(x) + x[::-1]
array([ 1.,  2.,  3.,  4.])
```

Large problem

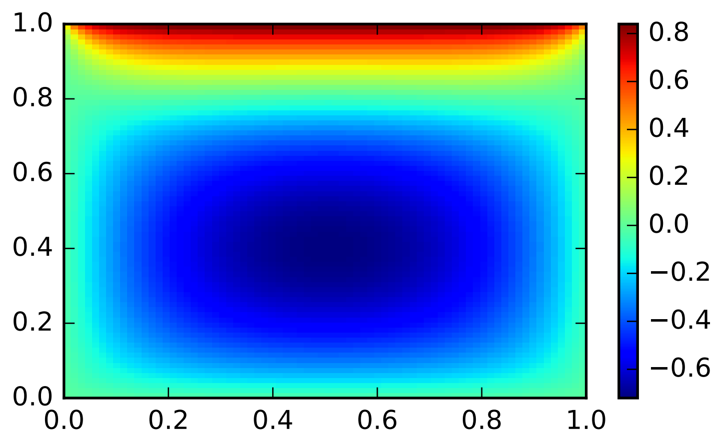
Suppose that we needed to solve the following integrodifferential equation on the square $[0, 1] \times [0, 1]$:

$$\nabla^2 P = 10 \left(\int_0^1 \int_0^1 \cosh(P) \, dx \, dy \right)^2$$

with $P(x, 1) = 1$ and $P = 0$ elsewhere on the boundary of the square.

The solution can be found using the *newton_krylov* solver:

```
import numpy as np
from scipy.optimize import newton_krylov
from numpy import cosh, zeros_like, mgrid, zeros

# parameters
nx, ny = 75, 75
hx, hy = 1./(nx-1), 1./(ny-1)

P_left, P_right = 0, 0
P_top, P_bottom = 1, 0

def residual(P):
    d2x = zeros_like(P)
    d2y = zeros_like(P)

    d2x[1:-1] = (P[2:]    - 2*P[1:-1] + P[:-2]) / hx/hx
    d2x[0]    = (P[1]     - 2*P[0]    + P_left)/hx/hx
    d2x[-1]   = (P_right - 2*P[-1]    + P[-2])/hx/hx

    d2y[:,1:-1] = (P[:,2:] - 2*P[:,1:-1] + P[:,:-2])/hy/hy
    d2y[:,0]    = (P[:,1]  - 2*P[:,0]    + P_bottom)/hy/hy
    d2y[:,-1]   = (P_top   - 2*P[:,-1]   + P[:,-2])/hy/hy

    return d2x + d2y - 10*cosh(P).mean()**2

# solve
guess = zeros((nx, ny), float)
sol = newton_krylov(residual, guess, method='lgmres', verbose=1)
print('Residual: %g' % abs(residual(sol)).max())

# visualize
import matplotlib.pyplot as plt
x, y = mgrid[0:1:(nx*1j), 0:1:(ny*1j)]
plt.pcolor(x, y, sol)
plt.colorbar()
plt.show()
```



5.20 Signal processing (`scipy.signal`)

5.20.1 Convolution

convolve(in1, in2[, mode, method])	Convolve two N-dimensional arrays.
correlate(in1, in2[, mode, method])	Cross-correlate two N-dimensional arrays.
fftconvolve(in1, in2[, mode])	Convolve two N-dimensional arrays using FFT.
convolve2d(in1, in2[, mode, boundary, fillvalue])	Convolve two 2-dimensional arrays.
correlate2d(in1, in2[, mode, boundary, ...])	Cross-correlate two 2-dimensional arrays.
sepfir2d((input, hrow, hcol) -> output)	Description:
choose_conv_method(in1, in2[, mode, measure])	Find the fastest convolution/correlation method.

scipy.signal.**convolve**(*in1*, *in2*, *mode='full'*, *method='auto'*)

Convolve two N-dimensional arrays.

Convolve *in1* and *in2*, with the output size determined by the *mode* argument.

Parameters

in1 : array_like
First input.

in2 : array_like
Second input. Should have the same number of dimensions as *in1*.

mode : str {'full', 'valid', 'same'}, optional
A string indicating the size of the output:

- ***full*** The output is the full discrete linear convolution of the inputs. (Default)
- ***valid*** The output consists only of those elements that do not rely on the zero-padding. In 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.
- ***same*** The output is the same size as *in1*, centered with respect to the 'full' output.

method : str {'auto', 'direct', 'fft'}, optional
A string indicating which method to use to calculate the convolution.

- ***direct*** The convolution is determined directly from sums, the definition of convolution.
- ***fft*** The Fourier Transform is used to perform the convolution by calling *fftconvolve*.
- ***auto*** Automatically chooses direct or Fourier method based on an estimate of which is faster (default). See Notes for more detail.

New in version 0.19.0.

Returns

convolve : array
An N-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

See also:

numpy.polymul
performs polynomial multiplication (same operation, but also accepts poly1d objects)

choose_conv_method
chooses the fastest appropriate convolution method

fftconvolve

Notes

By default, *convolve* and *correlate* use `method='auto'`, which calls *choose_conv_method* to choose the fastest method using pre-computed values (*choose_conv_method* can also measure real-world

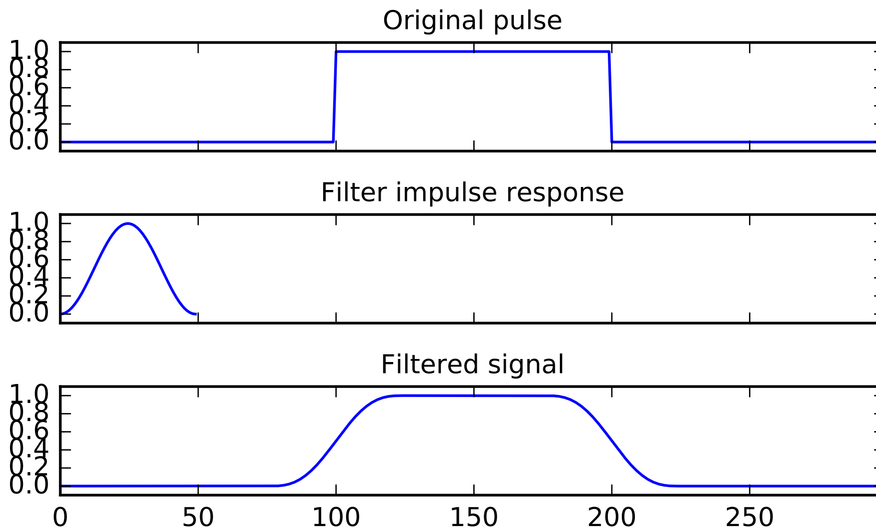
timing with a keyword argument). Because `fftconvolve` relies on floating point numbers, there are certain constraints that may force *method=direct* (more detail in `choose_conv_method` docstring).

Examples

Smooth a square pulse using a Hann window:

```
>>> from scipy import signal
>>> sig = np.repeat([0., 1., 0.], 100)
>>> win = signal.hann(50)
>>> filtered = signal.convolve(sig, win, mode='same') / sum(win)
```

```
>>> import matplotlib.pyplot as plt
>>> fig, (ax_orig, ax_win, ax_filt) = plt.subplots(3, 1, sharex=True)
>>> ax_orig.plot(sig)
>>> ax_orig.set_title('Original pulse')
>>> ax_orig.margins(0, 0.1)
>>> ax_win.plot(win)
>>> ax_win.set_title('Filter impulse response')
>>> ax_win.margins(0, 0.1)
>>> ax_filt.plot(filtered)
>>> ax_filt.set_title('Filtered signal')
>>> ax_filt.margins(0, 0.1)
>>> fig.tight_layout()
>>> fig.show()
```



`scipy.signal.correlate`(*in1*, *in2*, *mode='full'*, *method='auto'*)

Cross-correlate two N-dimensional arrays.

Cross-correlate *in1* and *in2*, with the output size determined by the *mode* argument.

Parameters

in1 : array_like
First input.

in2 : array_like
Second input. Should have the same number of dimensions as *in1*.

mode : str {'full', 'valid', 'same'}, optional
A string indicating the size of the output:

full The output is the full discrete linear cross-correlation of the inputs. (Default)

valid The output consists only of those elements that do not rely on the zero-padding. In 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

same The output is the same size as *in1*, centered with respect to the 'full' output.

method : str {'auto', 'direct', 'fft'}, optional

A string indicating which method to use to calculate the correlation.

direct The correlation is determined directly from sums, the definition of correlation.

fft The Fast Fourier Transform is used to perform the correlation more quickly (only available for numerical arrays.)

auto Automatically chooses direct or Fourier method based on an estimate of which is faster (default). See `convolve` Notes for more detail.

New in version 0.19.0.

Returns **correlate** : array

An N-dimensional array containing a subset of the discrete linear cross-correlation of *in1* with *in2*.

See also:

choose_conv_method
contains more documentation on *method*.

Notes

The correlation z of two d-dimensional arrays x and y is defined as:

```
z[...,k,...] = sum[..., i_l, ...] x[..., i_l,...] * conj(y[..., i_l - k,...])
```

This way, if x and y are 1-D arrays and `z = correlate(x, y, 'full')` then

$$z[k] = (x * y)(k - N + 1) = \sum_{l=0}^{||x||-1} x_l y^*_{l-k+N-1}$$

for $k = 0, 1, ..., ||x|| + ||y|| - 2$

where $||x||$ is the length of x, $N = \max(||x||, ||y||)$, and y_m is 0 when m is outside the range of y.

`method='fft'` only works for numerical arrays as it relies on `fftconvolve`. In certain cases (i.e., arrays of objects or when rounding integers can lose precision), `method='direct'` is always used.

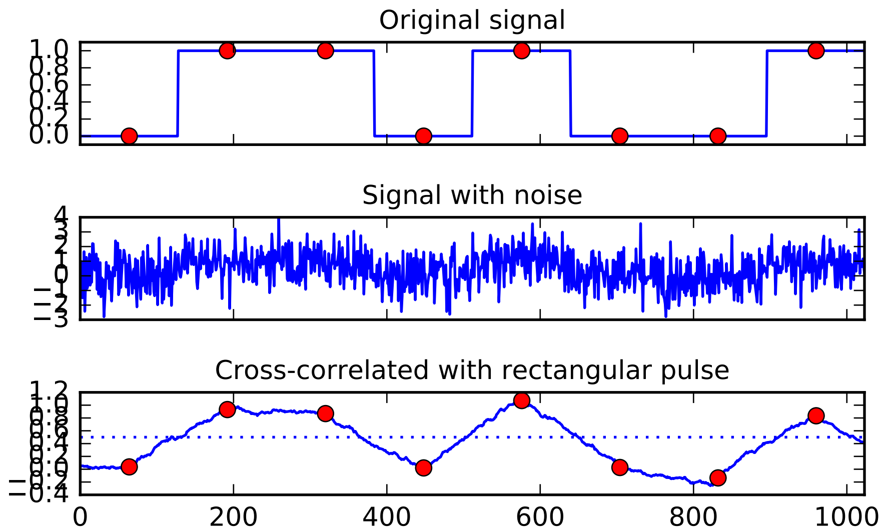
Examples

Implement a matched filter using cross-correlation, to recover a signal that has passed through a noisy channel.

```
>>> from scipy import signal
>>> sig = np.repeat([0., 1., 1., 0., 1., 0., 0., 1.], 128)
>>> sig_noise = sig + np.random.randn(len(sig))
>>> corr = signal.correlate(sig_noise, np.ones(128), mode='same') / 128
```

```
>>> import matplotlib.pyplot as plt
>>> clock = np.arange(64, len(sig), 128)
>>> fig, (ax_orig, ax_noise, ax_corr) = plt.subplots(3, 1, sharex=True)
>>> ax_orig.plot(sig)
>>> ax_orig.plot(clock, sig[clock], 'ro')
>>> ax_orig.set_title('Original signal')
>>> ax_noise.plot(sig_noise)
```

```
>>> ax_noise.set_title('Signal with noise')
>>> ax_corr.plot(corr)
>>> ax_corr.plot(clock, corr[clock], 'ro')
>>> ax_corr.axhline(0.5, ls=':')
>>> ax_corr.set_title('Cross-correlated with rectangular pulse')
>>> ax_orig.margins(0, 0.1)
>>> fig.tight_layout()
>>> fig.show()
```



`scipy.signal.fftconvolve(in1, in2, mode='full')`

Convolve two N-dimensional arrays using FFT.

Convolve *in1* and *in2* using the fast Fourier transform method, with the output size determined by the *mode* argument.

This is generally much faster than `convolve` for large arrays (n > ~500), but can be slower when only a few output values are needed, and can only output float arrays (int or object array inputs will be cast to float).

As of v0.19, `convolve` automatically chooses this method or the direct method based on an estimation of which is faster.

Parameters **in1** : array_like

First input.

in2 : array_like

Second input. Should have the same number of dimensions as *in1*. If operating in 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

mode : str {'full', 'valid', 'same'}, optional

A string indicating the size of the output:

full The output is the full discrete linear convolution of the inputs. (Default)

valid The output consists only of those elements that do not rely on the zero-padding.

same The output is the same size as *in1*, centered with respect to the 'full' output.

Returns **out** : array

An N-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

Examples

Autocorrelation of white noise is an impulse.

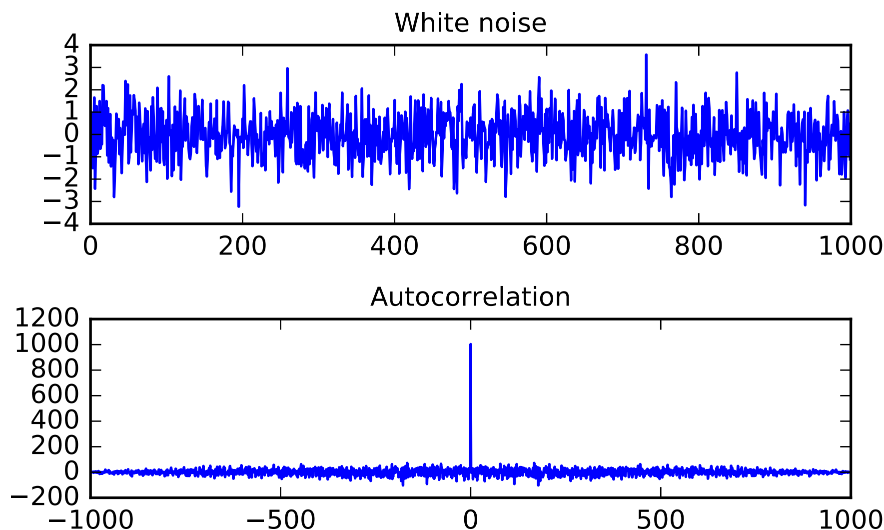
```
>>> from scipy import signal
>>> sig = np.random.randn(1000)
>>> autocorr = signal.fftconvolve(sig, sig[::-1], mode='full')
```

```
>>> import matplotlib.pyplot as plt
>>> fig, (ax_orig, ax_mag) = plt.subplots(2, 1)
>>> ax_orig.plot(sig)
>>> ax_orig.set_title('White noise')
>>> ax_mag.plot(np.arange(-len(sig)+1,len(sig)), autocorr)
>>> ax_mag.set_title('Autocorrelation')
>>> fig.tight_layout()
>>> fig.show()
```

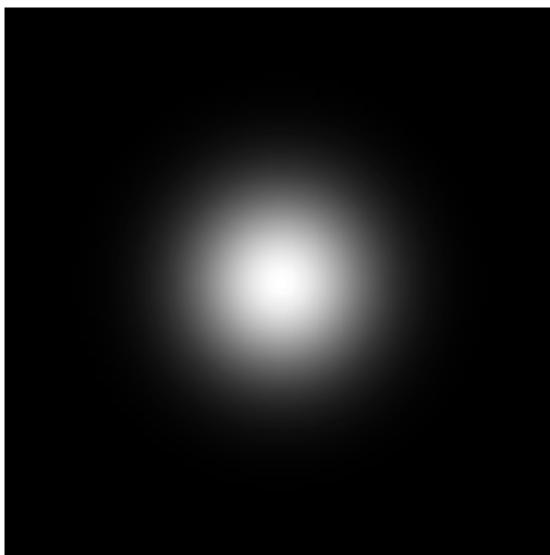
Gaussian blur implemented using FFT convolution. Notice the dark borders around the image, due to the zero-padding beyond its boundaries. The *convolve2d* function allows for other types of image boundaries, but is far slower.

```
>>> from scipy import misc
>>> face = misc.face(gray=True)
>>> kernel = np.outer(signal.gaussian(70, 8), signal.gaussian(70, 8))
>>> blurred = signal.fftconvolve(face, kernel, mode='same')
```

```
>>> fig, (ax_orig, ax_kernel, ax_blurred) = plt.subplots(3, 1,
...                                                      figsize=(6, 15))
>>> ax_orig.imshow(face, cmap='gray')
>>> ax_orig.set_title('Original')
>>> ax_orig.set_axis_off()
>>> ax_kernel.imshow(kernel, cmap='gray')
>>> ax_kernel.set_title('Gaussian kernel')
>>> ax_kernel.set_axis_off()
>>> ax_blurred.imshow(blurred, cmap='gray')
>>> ax_blurred.set_title('Blurred')
>>> ax_blurred.set_axis_off()
>>> fig.show()
```



Original


Gaussian kernel

Blurred

`scipy.signal.convolve2d`(*in1*, *in2*, *mode='full'*, *boundary='fill'*, *fillvalue=0*)

Convolve two 2-dimensional arrays.

Convolve *in1* and *in2* with output size determined by *mode*, and boundary conditions determined by *boundary* and *fillvalue*.

Parameters

in1 : array_like
First input.

in2 : array_like
Second input. Should have the same number of dimensions as *in1*. If operating in 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

mode : str {'full', 'valid', 'same'}, optional
A string indicating the size of the output:

full	The output is the full discrete linear convolution of the inputs. (Default)
valid	The output consists only of those elements that do not rely on the zero-padding.
same	The output is the same size as *in1*, centered with respect to the 'full' output.

boundary : str {'fill', 'wrap', 'symm'}, optional
A flag indicating how to handle boundaries:

fill	pad input arrays with fillvalue. (default)
wrap	circular boundary conditions.
symm	symmetrical boundary conditions.

fillvalue : scalar, optional
Value to fill pad input arrays with. Default is 0.

Returns

out : ndarray
A 2-dimensional array containing a subset of the discrete linear convolution of *in1* with *in2*.

Examples

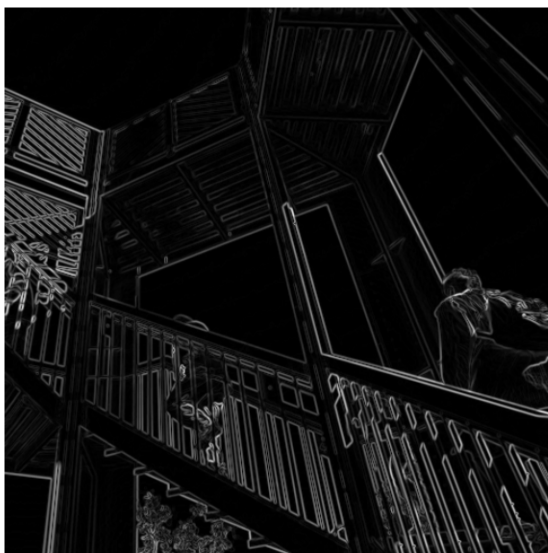
Compute the gradient of an image by 2D convolution with a complex Scharr operator. (Horizontal operator is real, vertical is imaginary.) Use symmetric boundary condition to avoid creating edges at the image boundaries.

```
>>> from scipy import signal
>>> from scipy import misc
>>> ascent = misc.ascent()
>>> scharr = np.array([[ -3-3j, 0-10j,  +3 -3j],
...                    [-10+0j, 0+ 0j, +10 +0j],
...                    [ -3+3j, 0+10j,  +3 +3j]]) # Gx + j*Gy
>>> grad = signal.convolve2d(ascent, scharr, boundary='symm', mode='same')
```

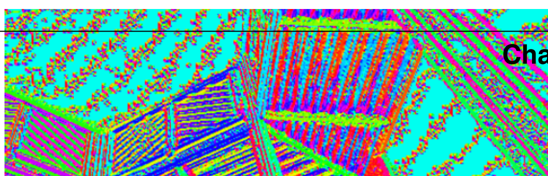
```
>>> import matplotlib.pyplot as plt
>>> fig, (ax_orig, ax_mag, ax_ang) = plt.subplots(3, 1, figsize=(6, 15))
>>> ax_orig.imshow(ascent, cmap='gray')
>>> ax_orig.set_title('Original')
>>> ax_orig.set_axis_off()
>>> ax_mag.imshow(np.absolute(grad), cmap='gray')
>>> ax_mag.set_title('Gradient magnitude')
>>> ax_mag.set_axis_off()
>>> ax_ang.imshow(np.angle(grad), cmap='hsv') # hsv is cyclic, like angles
>>> ax_ang.set_title('Gradient orientation')
>>> ax_ang.set_axis_off()
>>> fig.show()
```

Original

Gradient magnitude



Gradient orientation



scipy.signal.**correlate2d**(*in1*, *in2*, *mode='full'*, *boundary='fill'*, *fillvalue=0*)

Cross-correlate two 2-dimensional arrays.

Cross correlate *in1* and *in2* with output size determined by *mode*, and boundary conditions determined by *boundary* and *fillvalue*.

Parameters

in1 : array_like
First input.

in2 : array_like
Second input. Should have the same number of dimensions as *in1*. If operating in 'valid' mode, either *in1* or *in2* must be at least as large as the other in every dimension.

mode : str {'full', 'valid', 'same'}, optional
A string indicating the size of the output:

- ***full*** The output is the full discrete linear cross-correlation of the inputs. (Default)
- ***valid*** The output consists only of those elements that do not rely on the zero-padding.
- ***same*** The output is the same size as *in1*, centered with respect to the 'full' output.

boundary : str {'fill', 'wrap', 'symm'}, optional
A flag indicating how to handle boundaries:

- ***fill*** pad input arrays with fillvalue. (default)
- ***wrap*** circular boundary conditions.
- ***symm*** symmetrical boundary conditions.

fillvalue : scalar, optional
Value to fill pad input arrays with. Default is 0.

Returns

correlate2d : ndarray
A 2-dimensional array containing a subset of the discrete linear cross-correlation of *in1* with *in2*.

Examples

Use 2D cross-correlation to find the location of a template in a noisy image:

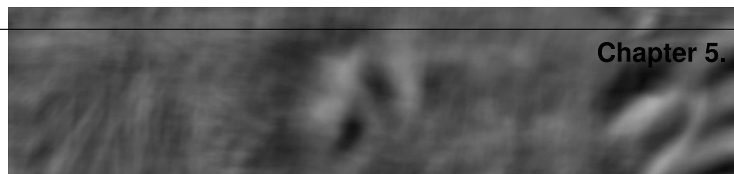
```
>>> from scipy import signal
>>> from scipy import misc
>>> face = misc.face(gray=True) - misc.face(gray=True).mean()
>>> template = np.copy(face[300:365, 670:750])  # right eye
>>> template -= template.mean()
>>> face = face + np.random.randn(*face.shape) * 50  # add noise
>>> corr = signal.correlate2d(face, template, boundary='symm', mode='same')
>>> y, x = np.unravel_index(np.argmax(corr), corr.shape)  # find the match
```

```
>>> import matplotlib.pyplot as plt
>>> fig, (ax_orig, ax_template, ax_corr) = plt.subplots(3, 1,
...                                                     figsize=(6, 15))
>>> ax_orig.imshow(face, cmap='gray')
>>> ax_orig.set_title('Original')
>>> ax_orig.set_axis_off()
>>> ax_template.imshow(template, cmap='gray')
>>> ax_template.set_title('Template')
>>> ax_template.set_axis_off()
>>> ax_corr.imshow(corr, cmap='gray')
>>> ax_corr.set_title('Cross-correlation')
>>> ax_corr.set_axis_off()
>>> ax_orig.plot(x, y, 'ro')
>>> fig.show()
```

Original

Template

Cross-correlation



`scipy.signal.sepfir2d(input, hrow, hcol)` → output

Description:

Convolve the rank-2 input array with the separable filter defined by the rank-1 arrays hrow, and hcol. Mirror symmetric boundary conditions are assumed. This function can be used to find an image given its B-spline representation.

`scipy.signal.choose_conv_method(in1, in2, mode='full', measure=False)`

Find the fastest convolution/correlation method.

This primarily exists to be called during the `method='auto'` option in *convolve* and *correlate*, but can also be used when performing many convolutions of the same input shapes and dtypes, determining which method to use for all of them, either to avoid the overhead of the 'auto' option or to use accurate real-world measurements.

Parameters

in1 : array_like

The first argument passed into the convolution function.

in2 : array_like

The second argument passed into the convolution function.

mode : str {'full', 'valid', 'same'}, optional

A string indicating the size of the output:

full The output is the full discrete linear convolution of the inputs. (Default)

valid The output consists only of those elements that do not rely on the zero-padding.

same The output is the same size as *in1*, centered with respect to the 'full' output.

measure : bool, optional

If True, run and time the convolution of *in1* and *in2* with both methods and return the fastest. If False (default), predict the fastest method using precomputed values.

Returns

method : str

A string indicating which convolution method is fastest, either 'direct' or 'fft'

times : dict, optional

A dictionary containing the times (in seconds) needed for each method. This value is only returned if `measure=True`.

See also:

convolve, *correlate*

Notes

For large n, `measure=False` is accurate and can quickly determine the fastest method to perform the convolution. However, this is not as accurate for small n (when any dimension in the input or output is small).

In practice, we found that this function estimates the faster method up to a multiplicative factor of 5 (i.e., the estimated method is *at most* 5 times slower than the fastest method). The estimation values were tuned on an early 2015 MacBook Pro with 8GB RAM but we found that the prediction held *fairly* accurately across different machines.

If `measure=True`, time the convolutions. Because this function uses *fftconvolve*, an error will be thrown if it does not support the inputs. There are cases when *fftconvolve* supports the inputs but this function returns *direct* (e.g., to protect against floating point integer precision).

New in version 0.19.

Examples

Estimate the fastest method for a given input:

```
>>> from scipy import signal
>>> a = np.random.randn(1000)
>>> b = np.random.randn(1000000)
>>> method = signal.choose_conv_method(a, b, mode='same')
>>> method
'fft'
```

This can then be applied to other arrays of the same dtype and shape:

```
>>> c = np.random.randn(1000)
>>> d = np.random.randn(1000000)
>>> # `method` works with correlate and convolve
>>> corr1 = signal.correlate(a, b, mode='same', method=method)
>>> corr2 = signal.correlate(c, d, mode='same', method=method)
>>> conv1 = signal.convolve(a, b, mode='same', method=method)
>>> conv2 = signal.convolve(c, d, mode='same', method=method)
```

5.20.2 B-splines

bspline(x, n)	B-spline basis function of order n.
cubic(x)	A cubic B-spline.
quadratic(x)	A quadratic B-spline.
gauss_spline(x, n)	Gaussian approximation to B-spline basis function of order n.
cspline1d(signal[, lamb])	Compute cubic spline coefficients for rank-1 array.
qspline1d(signal[, lamb])	Compute quadratic spline coefficients for rank-1 array.
cspline2d((input {, lambda, precision}) -> ck)	Description:
qspline2d((input {, lambda, precision}) -> qk)	Description:
cspline1d_eval(cj, newx[, dx, x0])	Evaluate a spline at the new set of points.
qspline1d_eval(cj, newx[, dx, x0])	Evaluate a quadratic spline at the new set of points.
spline_filter(Iin[, lmbda])	Smoothing spline (cubic) filtering of a rank-2 array.

scipy.signal.**bspline**(*x*, *n*)

B-spline basis function of order n.

Notes

Uses numpy.piecewise and automatic function-generator.

scipy.signal.**cubic**(*x*)

A cubic B-spline.

This is a special case of *bspline*, and equivalent to `bspline(x, 3)`.

scipy.signal.**quadratic**(*x*)

A quadratic B-spline.

This is a special case of *bspline*, and equivalent to `bspline(x, 2)`.

scipy.signal.**gauss_spline**(*x*, *n*)

Gaussian approximation to B-spline basis function of order n.

scipy.signal.**cspline1d**(*signal*, *lamb=0.0*)

Compute cubic spline coefficients for rank-1 array.

Find the cubic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window [1.0, 4.0, 1.0]/ 6.0 .

Parameters **signal** : ndarray
    A rank-1 array representing samples of a signal.
lamb : float, optional
    Smoothing coefficient, default is 0.0.

Returns **c** : ndarray
    Cubic spline coefficients.

`scipy.signal.qspline1d`(*signal*, *lamb=0.0*)

Compute quadratic spline coefficients for rank-1 array.

Find the quadratic spline coefficients for a 1-D signal assuming mirror-symmetric boundary conditions. To obtain the signal back from the spline representation mirror-symmetric-convolve these coefficients with a length 3 FIR window [1.0, 6.0, 1.0]/ 8.0 .

Parameters **signal** : ndarray
    A rank-1 array representing samples of a signal.
lamb : float, optional
    Smoothing coefficient (must be zero for now).

Returns **c** : ndarray
    Cubic spline coefficients.

`scipy.signal.cspline2d`(*input {*, *lambda*, *precision}*) → ck

Description:

Return the third-order B-spline coefficients over a regularly spacedi input grid for the two-dimensional input image. The lambda argument specifies the amount of smoothing. The precision argument allows specifying the precision used when computing the infinite sum needed to apply mirror- symmetric boundary conditions.

`scipy.signal.qspline2d`(*input {*, *lambda*, *precision}*) → qk

Description:

Return the second-order B-spline coefficients over a regularly spaced input grid for the two-dimensional input image. The lambda argument specifies the amount of smoothing. The precision argument allows specifying the precision used when computing the infinite sum needed to apply mirror- symmetric boundary conditions.

`scipy.signal.cspline1d_eval`(*cj*, *newx*, *dx=1.0*, *x0=0*)

Evaluate a spline at the new set of points.

dx is the old sample-spacing while *x0* was the old origin. In other-words the old-sample points (knot-points) for which the *cj* represent spline coefficients were at equally-spaced points of:

oldx = x0 + j*dx j=0...N-1, with N=len(cj)

Edges are handled using mirror-symmetric boundary conditions.

`scipy.signal.qspline1d_eval`(*cj*, *newx*, *dx=1.0*, *x0=0*)

Evaluate a quadratic spline at the new set of points.

dx is the old sample-spacing while *x0* was the old origin. In other-words the old-sample points (knot-points) for which the *cj* represent spline coefficients were at equally-spaced points of:

```
oldx = x0 + j*dx  j=0...N-1, with N=len(cj)
```

Edges are handled using mirror-symmetric boundary conditions.

scipy.signal.**spline_filter**(*Iin*, *lmbda=5.0*)
Smoothing spline (cubic) filtering of a rank-2 array.

Filter an input data set, *Iin*, using a (cubic) smoothing spline of fall-off *lmbda*.

5.20.3 Filtering

order_filter(a, domain, rank)	Perform an order filter on an N-dimensional array.
medfilt(volume[, kernel_size])	Perform a median filter on an N-dimensional array.
medfilt2d(input[, kernel_size])	Median filter a 2-dimensional array.
wiener(im[, mysize, noise])	Perform a Wiener filter on an N-dimensional array.
symiirorder1((input, c0, z1 {, ...)	Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of first-order sections.
symiirorder2((input, r, omega {, ...)	Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of second-order sections.
lfilter(b, a, x[, axis, zi])	Filter data along one-dimension with an IIR or FIR filter.
lfiltic(b, a, y[, x])	Construct initial conditions for lfilter.
lfilter_zi(b, a)	Compute an initial state *zi* for the lfilter function that corresponds to the steady state of the step response.
filtfilt(b, a, x[, axis, padtype, padlen, ...])	A forward-backward filter.
savgol_filter(x, window_length, polyorder[, ...])	Apply a Savitzky-Golay filter to an array.
deconvolve(signal, divisor)	Deconvolves `divisor` out of `signal`.
sosfilt(sos, x[, axis, zi])	Filter data along one dimension using cascaded second-order sections
sosfilt_zi(sos)	Compute an initial state *zi* for the sosfilt function that corresponds to the steady state of the step response.
sosfiltfilt(sos, x[, axis, padtype, padlen])	A forward-backward filter using cascaded second-order sections.
hilbert(x[, N, axis])	Compute the analytic signal, using the Hilbert transform.
hilbert2(x[, N])	Compute the '2-D' analytic signal of *x*
decimate(x, q[, n, ftype, axis, zero_phase])	Downsample the signal after applying an anti-aliasing filter.
detrend(data[, axis, type, bp])	Remove linear trend along axis from data.
resample(x, num[, t, axis, window])	Resample *x* to *num* samples using Fourier method along the given axis.
resample_poly(x, up, down[, axis, window])	Resample *x* along the given axis using polyphase filtering.
upfirdn(h, x[, up, down, axis])	Upsample, FIR filter, and downsample

scipy.signal.**order_filter**(*a*, *domain*, *rank*)
Perform an order filter on an N-dimensional array.

Perform an order filter on the array in. The domain argument acts as a mask centered over each pixel. The non-zero elements of domain are used to select elements surrounding each input pixel which are placed in a list. The list is sorted, and the output for that pixel is the element corresponding to rank in the sorted list.

Parameters
a : ndarray
The N-dimensional input array.

domain : array_like
A mask array with the same number of dimensions as *a*. Each dimension should have an odd number of elements.

rank : int

A non-negative integer which selects the element from the sorted list (0 corresponds to the smallest element, 1 is the next smallest element, etc.).

Returns **out** : ndarray

The results of the order filter in an array with the same shape as *a*.

Examples

```
>>> from scipy import signal
>>> x = np.arange(25).reshape(5, 5)
>>> domain = np.identity(3)
>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> signal.order_filter(x, domain, 0)
array([[  0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   1.,   2.,   0.],
       [  0.,   5.,   6.,   7.,   0.],
       [  0.,  10.,  11.,  12.,   0.],
       [  0.,   0.,   0.,   0.,   0.]])
>>> signal.order_filter(x, domain, 2)
array([[  6.,   7.,   8.,   9.,   4.],
       [ 11.,  12.,  13.,  14.,   9.],
       [ 16.,  17.,  18.,  19.,  14.],
       [ 21.,  22.,  23.,  24.,  19.],
       [ 20.,  21.,  22.,  23.,  24.]])
```

`scipy.signal.`**`medfilt`**(*volume*, *kernel_size=None*)

Perform a median filter on an N-dimensional array.

Apply a median filter to the input array using a local window-size given by *kernel_size*.

Parameters **volume** : array_like

An N-dimensional input array.

kernel_size : array_like, optional

A scalar or an N-length list giving the size of the median filter window in each dimension. Elements of *kernel_size* should be odd. If *kernel_size* is a scalar, then this scalar is used as the size in each dimension. Default size is 3 for each dimension.

Returns **out** : ndarray

An array the same size as input containing the median filtered result.

`scipy.signal.`**`medfilt2d`**(*input*, *kernel_size=3*)

Median filter a 2-dimensional array.

Apply a median filter to the *input* array using a local window-size given by *kernel_size* (must be odd).

Parameters **input** : array_like

A 2-dimensional input array.

kernel_size : array_like, optional

A scalar or a list of length 2, giving the size of the median filter window in each dimension. Elements of *kernel_size* should be odd. If *kernel_size* is a scalar, then this scalar is used as the size in each dimension. Default is a kernel of size (3, 3).

Returns **out** : ndarray

An array the same size as input containing the median filtered result.

`scipy.signal.`**`wiener`**(*im*, *mysize=None*, *noise=None*)

Perform a Wiener filter on an N-dimensional array.

Apply a Wiener filter to the N-dimensional array *im*.

Parameters

im : ndarray
: An N-dimensional array.

mysize : int or array_like, optional
: A scalar or an N-length list giving the size of the Wiener filter window in each dimension. Elements of mysize should be odd. If mysize is a scalar, then this scalar is used as the size in each dimension.

noise : float, optional
: The noise-power to use. If None, then noise is estimated as the average of the local variance of the input.

Returns

out : ndarray
: Wiener filtered result with the same shape as *im*.

scipy.signal.**symiirorder1**(*input*, *c0*, *z1 {*, *precision}*) → output

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of first-order sections. The second section uses a reversed sequence. This implements a system with the following transfer function and mirror-symmetric boundary conditions:

```
                c0
H(z) = ---------------------
        (1-z1/z) (1 - z1 z)
```

The resulting signal will have mirror symmetric boundary conditions as well.

Parameters

input : ndarray
: The input signal.

c0, z1 : scalar
: Parameters in the transfer function.

precision :
: Specifies the precision for calculating initial conditions of the recursive filter based on mirror-symmetric input.

Returns

output : ndarray
: The filtered signal.

scipy.signal.**symiirorder2**(*input*, *r*, *omega {*, *precision}*) → output

Implement a smoothing IIR filter with mirror-symmetric boundary conditions using a cascade of second-order sections. The second section uses a reversed sequence. This implements the following transfer function:

```
                         cs^2
H(z) = ---------------------------------------
       (1 - a2/z - a3/z^2) (1 - a2 z - a3 z^2 )
```

where:

```
a2 = (2 r cos omega)
a3 = - r^2
cs = 1 - 2 r cos omega + r^2
```

Parameters

input : ndarray
: The input signal.

r, omega : scalar
: Parameters in the transfer function.

precision :
: Specifies the precision for calculating initial conditions of the recursive filter based on mirror-symmetric input.

Returns

output : ndarray
: The filtered signal.

`scipy.signal.lfilter(b, a, x, axis=-1, zi=None)`

Filter data along one-dimension with an IIR or FIR filter.

Filter a data sequence, *x*, using a digital filter. This works for many fundamental data types (including Object type). The filter is a direct form II transposed implementation of the standard difference equation (see Notes).

Parameters

b : array_like
: The numerator coefficient vector in a 1-D sequence.

a : array_like
: The denominator coefficient vector in a 1-D sequence. If `a[0]` is not 1, then both *a* and *b* are normalized by `a[0]`.

x : array_like
: An N-dimensional input array.

axis : int, optional
: The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.

zi : array_like, optional
: Initial conditions for the filter delays. It is a vector (or array of vectors for an N-dimensional input) of length `max(len(a), len(b)) - 1`. If *zi* is None or is not given then initial rest is assumed. See `lfiltic` for more information.

Returns

y : array
: The output of the digital filter.

zf : array, optional
: If *zi* is None, this is not returned, otherwise, *zf* holds the final filter delay values.

See also:

`lfiltic` Construct initial conditions for `lfilter`.
`lfilter_zi` Compute initial state (steady state of step response) for `lfilter`.
`filtfilt` A forward-backward filter, to obtain a filter with linear phase.
`savgol_filter` A Savitzky-Golay filter.
`sosfilt` Filter data using cascaded second-order sections.
`sosfiltfilt` A forward-backward filter using second-order sections.

Notes

The filter function is implemented as a direct II transposed structure. This means that the filter implements:

```
a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M]
                      - a[1]*y[n-1] - ... - a[N]*y[n-N]
```

where *M* is the degree of the numerator, *N* is the degree of the denominator, and *n* is the sample number. It is implemented using the following difference equations (assuming M = N):

```
a[0]*y[n] = b[0] * x[n]               + d[0][n-1]
  d[0][n] = b[1] * x[n] - a[1] * y[n] + d[1][n-1]
  d[1][n] = b[2] * x[n] - a[2] * y[n] + d[2][n-1]
...
d[N-2][n] = b[N-1]*x[n] - a[N-1]*y[n] + d[N-1][n-1]
d[N-1][n] = b[N] * x[n] - a[N] * y[n]
```

where *d* are the state variables.

The rational transfer function describing this filter in the z-transform domain is:

```
                    -1                -M
        b[0] + b[1]z  + ... + b[M] z
Y(z) = -------------------------------- X(z)
                    -1                -N
        a[0] + a[1]z  + ... + a[N] z
```

Examples

Generate a noisy signal to be filtered:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(-1, 1, 201)
>>> x = (np.sin(2*np.pi*0.75*t*(1-t) + 2.1) +
...      0.1*np.sin(2*np.pi*1.25*t + 1) +
...      0.18*np.cos(2*np.pi*3.85*t))
>>> xn = x + np.random.randn(len(t)) * 0.08
```

Create an order 3 lowpass butterworth filter:

```
>>> b, a = signal.butter(3, 0.05)
```

Apply the filter to xn. Use lfilter_zi to choose the initial condition of the filter:

```
>>> zi = signal.lfilter_zi(b, a)
>>> z, _ = signal.lfilter(b, a, xn, zi=zi*xn[0])
```

Apply the filter again, to have a result filtered at an order the same as filtfilt:

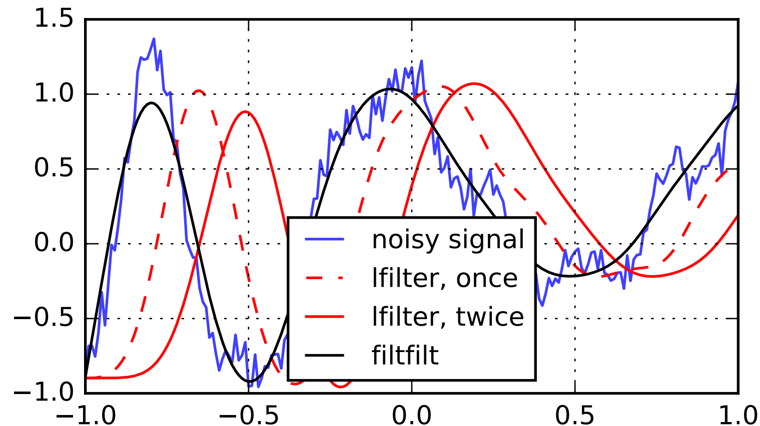
```
>>> z2, _ = signal.lfilter(b, a, z, zi=zi*z[0])
```

Use filtfilt to apply the filter:

```
>>> y = signal.filtfilt(b, a, xn)
```

Plot the original signal and the various filtered versions:

```
>>> plt.figure
>>> plt.plot(t, xn, 'b', alpha=0.75)
>>> plt.plot(t, z, 'r--', t, z2, 'r', t, y, 'k')
>>> plt.legend(('noisy signal', 'lfilter, once', 'lfilter, twice',
...             'filtfilt'), loc='best')
>>> plt.grid(True)
>>> plt.show()
```



`scipy.signal.`**`lfiltic`**(*b*, *a*, *y*, *x=None*)

Construct initial conditions for lfilter.

Given a linear filter (b, a) and initial conditions on the output *y* and the input *x*, return the initial conditions on the state vector zi which is used by *`lfilter`* to generate the output given the input.

Parameters

b : array_like

Linear filter term.

a : array_like

Linear filter term.

y : array_like

Initial conditions.

If `N = len(a) - 1`, then `y = {y[-1], y[-2], ..., y[-N]}`.

If *y* is too short, it is padded with zeros.

x : array_like, optional

Initial conditions.

If `M = len(b) - 1`, then `x = {x[-1], x[-2], ..., x[-M]}`.

If *x* is not given, its initial conditions are assumed zero.

If *x* is too short, it is padded with zeros.

Returns

zi : ndarray

The state vector `zi = {z_0[-1], z_1[-1], ..., z_K-1[-1]}`, where `K = max(M, N)`.

See also:

`lfilter`, *`lfilter_zi`*

`scipy.signal.`**`lfilter_zi`**(*b*, *a*)

Compute an initial state *zi* for the lfilter function that corresponds to the steady state of the step response.

A typical use of this function is to set the initial state so that the output of the filter starts at the same value as the first element of the signal to be filtered.

Parameters

b, a : array_like (1-D)

The IIR filter coefficients. See *`lfilter`* for more information.

Returns

zi : 1-D ndarray

The initial state for the filter.

See also:

`lfilter`, *`lfiltic`*, *`filtfilt`*

Notes

A linear filter with order m has a state space representation (A, B, C, D), for which the output y of the filter can be expressed as:

```
z(n+1) = A*z(n) + B*x(n)
y(n)   = C*z(n) + D*x(n)
```

where z(n) is a vector of length m, A has shape (m, m), B has shape (m, 1), C has shape (1, m) and D has shape (1, 1) (assuming x(n) is a scalar). lfilter_zi solves:

```
zi = A*zi + B
```

In other words, it finds the initial condition for which the response to an input of all ones is a constant.

Given the filter coefficients *a* and *b*, the state space matrices for the transposed direct form II implementation of the linear filter, which is the implementation used by scipy.signal.lfilter, are:

```
A = scipy.linalg.companion(a).T
B = b[1:] - a[1:]*b[0]
```

assuming *a[0]* is 1.0; if *a[0]* is not 1, *a* and *b* are first divided by a[0].

Examples

The following code creates a lowpass Butterworth filter. Then it applies that filter to an array whose values are all 1.0; the output is also all 1.0, as expected for a lowpass filter. If the *zi* argument of `lfilter` had not been given, the output would have shown the transient signal.

```
>>> from numpy import array, ones
>>> from scipy.signal import lfilter, lfilter_zi, butter
>>> b, a = butter(5, 0.25)
>>> zi = lfilter_zi(b, a)
>>> y, zo = lfilter(b, a, ones(10), zi=zi)
>>> y
array([1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Another example:

```
>>> x = array([0.5, 0.5, 0.5, 0.0, 0.0, 0.0, 0.0])
>>> y, zf = lfilter(b, a, x, zi=zi*x[0])
>>> y
array([ 0.5       ,  0.5       ,  0.5       ,  0.49836039,  0.48610528,
    0.44399389,  0.35505241])
```

Note that the *zi* argument to `lfilter` was computed using `lfilter_zi` and scaled by *x[0]*. Then the output *y* has no transient until the input drops from 0.5 to 0.0.

scipy.signal.**filtfilt**(*b*, *a*, *x*, *axis=-1*, *padtype='odd'*, *padlen=None*, *method='pad'*, *irlen=None*)

A forward-backward filter.

This function applies a linear filter twice, once forward and once backwards. The combined filter has linear phase.

The function provides options for handling the edges of the signal.

When *method* is "pad", the function pads the data along the given axis in one of three ways: odd, even or constant. The odd and even extensions have the corresponding symmetry about the end point of the data. The constant extension extends the data with the values at the end points. On both the forward and backward passes,

the initial condition of the filter is found by using `lfilter_zi` and scaling it by the end point of the extended data.

When *method* is "gust", Gustafsson's method *[R216]* is used. Initial conditions are chosen for the forward and backward passes so that the forward-backward filter gives the same result as the backward-forward filter.

Parameters

b : (N,) array_like
: The numerator coefficient vector of the filter.

a : (N,) array_like
: The denominator coefficient vector of the filter. If `a[0]` is not 1, then both *a* and *b* are normalized by `a[0]`.

x : array_like
: The array of data to be filtered.

axis : int, optional
: The axis of *x* to which the filter is applied. Default is -1.

padtype : str or None, optional
: Must be 'odd', 'even', 'constant', or None. This determines the type of extension to use for the padded signal to which the filter is applied. If *padtype* is None, no padding is used. The default is 'odd'.

padlen : int or None, optional
: The number of elements by which to extend *x* at both ends of *axis* before applying the filter. This value must be less than `x.shape[axis] - 1`. `padlen=0` implies no padding. The default value is `3 * max(len(a), len(b))`.

method : str, optional
: Determines the method for handling the edges of the signal, either "pad" or "gust". When *method* is "pad", the signal is padded; the type of padding is determined by *padtype* and *padlen*, and *irlen* is ignored. When *method* is "gust", Gustafsson's method is used, and *padtype* and *padlen* are ignored.

irlen : int or None, optional
: When *method* is "gust", *irlen* specifies the length of the impulse response of the filter. If *irlen* is None, no part of the impulse response is ignored. For a long signal, specifying *irlen* can significantly improve the performance of the filter.

Returns

y : ndarray
: The filtered output with the same shape as *x*.

See also:

`sosfiltfilt`, `lfilter_zi`, `lfilter`, `lfiltic`, `savgol_filter`, `sosfilt`

Notes

The option to use Gustaffson's method was added in scipy version 0.16.0.

References

[R216]

Examples

The examples will use several functions from `scipy.signal`.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

First we create a one second signal that is the sum of two pure sine waves, with frequencies 5 Hz and 250 Hz, sampled at 2000 Hz.

```
>>> t = np.linspace(0, 1.0, 2001)
>>> xlow = np.sin(2 * np.pi * 5 * t)
>>> xhigh = np.sin(2 * np.pi * 250 * t)
>>> x = xlow + xhigh
```

Now create a lowpass Butterworth filter with a cutoff of 0.125 times the Nyquist rate, or 125 Hz, and apply it to `x` with *filtfilt*. The result should be approximately `xlow`, with no phase shift.

```
>>> b, a = signal.butter(8, 0.125)
>>> y = signal.filtfilt(b, a, x, padlen=150)
>>> np.abs(y - xlow).max()
9.1086182074789912e-06
```

We get a fairly clean result for this artificial example because the odd extension is exact, and with the moderately long padding, the filter's transients have dissipated by the time the actual data is reached. In general, transient effects at the edges are unavoidable.

The following example demonstrates the option `method="gust"`.

First, create a filter.

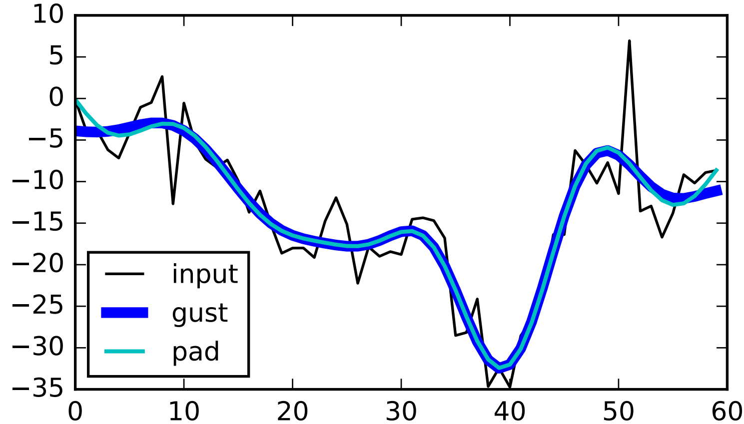
```
>>> b, a = signal.ellip(4, 0.01, 120, 0.125)  # Filter to be applied.
>>> np.random.seed(123456)
```

sig is a random input signal to be filtered.

```
>>> n = 60
>>> sig = np.random.randn(n)**3 + 3*np.random.randn(n).cumsum()
```

Apply *filtfilt* to *sig*, once using the Gustafsson method, and once using padding, and plot the results for comparison.

```
>>> fgust = signal.filtfilt(b, a, sig, method="gust")
>>> fpad = signal.filtfilt(b, a, sig, padlen=50)
>>> plt.plot(sig, 'k-', label='input')
>>> plt.plot(fgust, 'b-', linewidth=4, label='gust')
>>> plt.plot(fpad, 'c-', linewidth=1.5, label='pad')
>>> plt.legend(loc='best')
>>> plt.show()
```



The *irlen* argument can be used to improve the performance of Gustafsson's method.

Estimate the impulse response length of the filter.

```
>>> z, p, k = signal.tf2zpk(b, a)
>>> eps = 1e-9
>>> r = np.max(np.abs(p))
>>> approx_impulse_len = int(np.ceil(np.log(eps) / np.log(r)))
>>> approx_impulse_len
137
```

Apply the filter to a longer signal, with and without the *irlen* argument. The difference between *y1* and *y2* is small. For long signals, using *irlen* gives a significant performance improvement.

```
>>> x = np.random.randn(5000)
>>> y1 = signal.filtfilt(b, a, x, method='gust')
>>> y2 = signal.filtfilt(b, a, x, method='gust', irlen=approx_impulse_len)
>>> print(np.max(np.abs(y1 - y2)))
1.80056858312e-10
```

`scipy.signal.`**`savgol_filter`**(*x*, *window_length*, *polyorder*, *deriv=0*, *delta=1.0*, *axis=-1*, *mode='interp'*, *cval=0.0*)

Apply a Savitzky-Golay filter to an array.

This is a 1-d filter. If *x* has dimension greater than 1, *axis* determines the axis along which the filter is applied.

Parameters **x** : array_like

The data to be filtered. If *x* is not a single or double precision floating point array, it will be converted to type `numpy.float64` before filtering.

window_length : int

The length of the filter window (i.e. the number of coefficients). *window_length* must be a positive odd integer.

polyorder : int

The order of the polynomial used to fit the samples. *polyorder* must be less than *window_length*.

deriv : int, optional

The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.

delta : float, optional
The spacing of the samples to which the filter will be applied. This is only used if deriv > 0. Default is 1.0.

axis : int, optional
The axis of the array *x* along which the filter is to be applied. Default is -1.

mode : str, optional
Must be 'mirror', 'constant', 'nearest', 'wrap' or 'interp'. This determines the type of extension to use for the padded signal to which the filter is applied. When *mode* is 'constant', the padding value is given by *cval*. See the Notes for more details on 'mirror', 'constant', 'wrap', and 'nearest'. When the 'interp' mode is selected (the default), no extension is used. Instead, a degree *polyorder* polynomial is fit to the last *window_length* values of the edges, and this polynomial is used to evaluate the last *window_length // 2* output values.

cval : scalar, optional
Value to fill past the edges of the input if *mode* is 'constant'. Default is 0.0.

Returns **y** : ndarray, same shape as *x*
The filtered data.

See also:

`savgol_coeffs`

Notes

Details on the *mode* options:

'mirror': Repeats the values at the edges in reverse order. The value closest to the edge is not included.

'nearest': The extension contains the nearest input value.

'constant': The extension contains the value given by the *cval* argument.

'wrap': The extension contains the values from the other end of the array.

For example, if the input is [1, 2, 3, 4, 5, 6, 7, 8], and *window_length* is 7, the following shows the extended data for the various *mode* options (assuming *cval* is 0):

```
mode       |   Ext   |         Input          |   Ext
-----------+---------+------------------------+---------
'mirror'   | 4  3  2 | 1  2  3  4  5  6  7  8 | 7  6  5
'nearest'  | 1  1  1 | 1  2  3  4  5  6  7  8 | 8  8  8
'constant' | 0  0  0 | 1  2  3  4  5  6  7  8 | 0  0  0
'wrap'     | 6  7  8 | 1  2  3  4  5  6  7  8 | 1  2  3
```

New in version 0.14.0.

Examples

```
>>> from scipy.signal import savgol_filter
>>> np.set_printoptions(precision=2)  # For compact display.
>>> x = np.array([2, 2, 5, 2, 1, 0, 1, 4, 9])
```

Filter with a window length of 5 and a degree 2 polynomial. Use the defaults for all other parameters.

```
>>> savgol_filter(x, 5, 2)
array([ 1.66,  3.17,  3.54,  2.86,  0.66,  0.17,  1.  ,  4.  ,  9.  ])
```

Note that the last five values in x are samples of a parabola, so when mode='interp' (the default) is used with polyorder=2, the last three values are unchanged. Compare that to, for example, *mode='nearest'*:

```
>>> savgol_filter(x, 5, 2, mode='nearest')
array([ 1.74,  3.03,  3.54,  2.86,  0.66,  0.17,  1.  ,  4.6 ,  7.97])
```

`scipy.signal.deconvolve`(*signal*, *divisor*)

Deconvolves `divisor` out of `signal`.

Returns the quotient and remainder such that `signal = convolve(divisor, quotient) + remainder`

Parameters

signal : array_like
: Signal data, typically a recorded signal

divisor : array_like
: Divisor data, typically an impulse response or filter that was applied to the original signal

Returns

quotient : ndarray
: Quotient, typically the recovered original signal

remainder : ndarray
: Remainder

See also:

numpy.polydiv
: performs polynomial division (same operation, but also accepts poly1d objects)

Examples

Deconvolve a signal that's been filtered:

```
>>> from scipy import signal
>>> original = [0, 1, 0, 0, 1, 1, 0, 0]
>>> impulse_response = [2, 1]
>>> recorded = signal.convolve(impulse_response, original)
>>> recorded
array([0, 2, 1, 0, 2, 3, 1, 0, 0])
>>> recovered, remainder = signal.deconvolve(recorded, impulse_response)
>>> recovered
array([ 0.,  1.,  0.,  0.,  1.,  1.,  0.,  0.])
```

`scipy.signal.sosfilt`(*sos*, *x*, *axis=-1*, *zi=None*)

Filter data along one dimension using cascaded second-order sections

Filter a data sequence, *x*, using a digital IIR filter defined by *sos*. This is implemented by performing `lfilter` for each second-order section. See `lfilter` for details.

Parameters

sos : array_like
: Array of second-order filter coefficients, must have shape `(n_sections, 6)`. Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.

x : array_like
: An N-dimensional input array.

axis : int, optional
: The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.

zi : array_like, optional
: Initial conditions for the cascaded filter delays. It is a (at least 2D) vector of shape `(n_sections, ..., 2, ...)`, where `..., 2, ...` denotes the shape of *x*, but with `x.shape[axis]` replaced by 2. If *zi* is None or is not given then initial rest (i.e. all zeros) is assumed. Note that

these initial conditions are *not* the same as the initial conditions given by `lfiltic` or `lfilter_zi`.

Returns **y** : ndarray

The output of the digital filter.

zf : ndarray, optional

If *zi* is None, this is not returned, otherwise, *zf* holds the final filter delay values.

See also:

zpk2sos, *sos2zpk*, *sosfilt_zi*, *sosfiltfilt*, *sosfreqz*

Notes

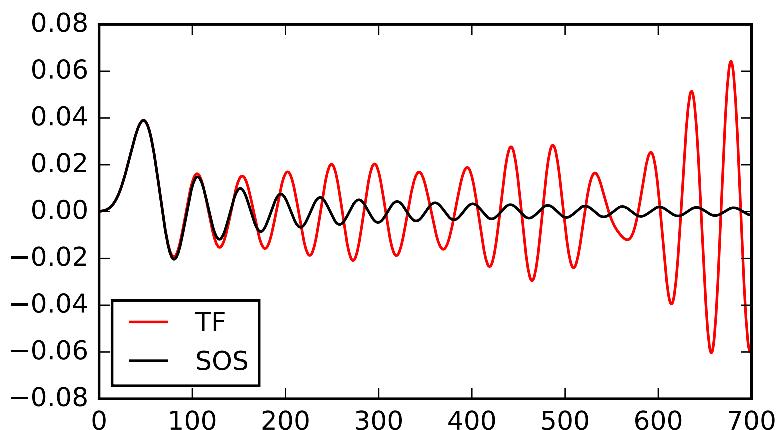
The filter function is implemented as a series of second-order filters with direct-form II transposed structure. It is designed to minimize numerical precision errors for high-order filters.

New in version 0.16.0.

Examples

Plot a 13th-order filter's impulse response using both `lfilter` and `sosfilt`, showing the instability that results from trying to do a 13th-order filter in a single stage (the numerical error pushes some poles outside of the unit circle):

```
>>> import matplotlib.pyplot as plt
>>> from scipy import signal
>>> b, a = signal.ellip(13, 0.009, 80, 0.05, output='ba')
>>> sos = signal.ellip(13, 0.009, 80, 0.05, output='sos')
>>> x = signal.unit_impulse(700)
>>> y_tf = signal.lfilter(b, a, x)
>>> y_sos = signal.sosfilt(sos, x)
>>> plt.plot(y_tf, 'r', label='TF')
>>> plt.plot(y_sos, 'k', label='SOS')
>>> plt.legend(loc='best')
>>> plt.show()
```



`scipy.signal.`**`sosfilt_zi`**(*sos*)

Compute an initial state *zi* for the sosfilt function that corresponds to the steady state of the step response.

A typical use of this function is to set the initial state so that the output of the filter starts at the same value as the first element of the signal to be filtered.

Parameters **sos** : array_like

Array of second-order filter coefficients, must have shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

Returns **zi** : ndarray

Initial conditions suitable for use with `sosfilt`, shape `(n_sections, 2)`.

See also:

sosfilt, *zpk2sos*

Notes

New in version 0.16.0.

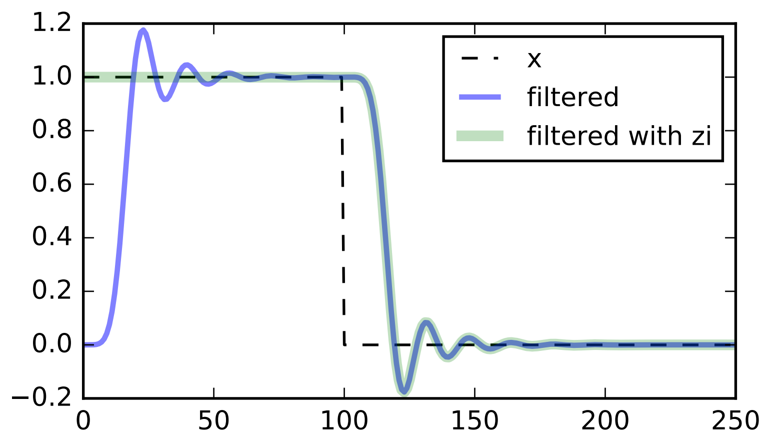
Examples

Filter a rectangular pulse that begins at time 0, with and without the use of the *zi* argument of *scipy.signal.sosfilt*.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> sos = signal.butter(9, 0.125, output='sos')
>>> zi = signal.sosfilt_zi(sos)
>>> x = (np.arange(250) < 100).astype(int)
>>> f1 = signal.sosfilt(sos, x)
>>> f2, zo = signal.sosfilt(sos, x, zi=zi)
```

```
>>> plt.plot(x, 'k--', label='x')
>>> plt.plot(f1, 'b', alpha=0.5, linewidth=2, label='filtered')
>>> plt.plot(f2, 'g', alpha=0.25, linewidth=4, label='filtered with zi')
>>> plt.legend(loc='best')
>>> plt.show()
```



`scipy.signal.sosfiltfilt`(*sos*, *x*, *axis=-1*, *padtype='odd'*, *padlen=None*)

A forward-backward filter using cascaded second-order sections.

See `filtfilt` for more complete information about this method.

Parameters

sos : array_like
: Array of second-order filter coefficients, must have shape `(n_sections, 6)`. Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.

x : array_like
: The array of data to be filtered.

axis : int, optional
: The axis of *x* to which the filter is applied. Default is -1.

padtype : str or None, optional
: Must be 'odd', 'even', 'constant', or None. This determines the type of extension to use for the padded signal to which the filter is applied. If *padtype* is None, no padding is used. The default is 'odd'.

padlen : int or None, optional
: The number of elements by which to extend *x* at both ends of *axis* before applying the filter. This value must be less than `x.shape[axis] - 1`. `padlen=0` implies no padding. The default value is:

```
3 * (2 * len(sos) + 1 - min((sos[:, 2] == 0).sum(),
                            (sos[:, 5] == 0).sum()))
```

The extra subtraction at the end attempts to compensate for poles and zeros at the origin (e.g. for odd-order filters) to yield equivalent estimates of *padlen* to those of `filtfilt` for second-order section filters built with `scipy.signal` functions.

Returns

y : ndarray
: The filtered output with the same shape as *x*.

See also:

`filtfilt`, `sosfilt`, `sosfilt_zi`, `sosfreqz`

Notes

New in version 0.18.0.

`scipy.signal.hilbert`(*x*, *N=None*, *axis=-1*)

Compute the analytic signal, using the Hilbert transform.

The transformation is done along the last axis by default.

Parameters

x : array_like
: Signal data. Must be real.

N : int, optional
: Number of Fourier components. Default: `x.shape[axis]`

axis : int, optional
: Axis along which to do the transformation. Default: -1.

Returns

xa : ndarray
: Analytic signal of *x*, of each 1-D array along *axis*

See also:

`scipy.fftpack.hilbert`
: Return Hilbert transform of a periodic sequence x.

Notes

The analytic signal `x_a(t)` of signal `x(t)` is:

$$x_a = F^{-1}(F(x)2U) = x + iy$$

where F is the Fourier transform, U the unit step function, and y the Hilbert transform of x. *[R235]*

In other words, the negative half of the frequency spectrum is zeroed out, turning the real-valued signal into a complex signal. The Hilbert transformed signal can be obtained from `np.imag(hilbert(x))`, and the original signal from `np.real(hilbert(x))`.

References

[R235], *[R236]*, *[R237]*

Examples

In this example we use the Hilbert transform to determine the amplitude envelope and instantaneous frequency of an amplitude-modulated signal.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.signal import hilbert, chirp
```

```
>>> duration = 1.0
>>> fs = 400.0
>>> samples = int(fs*duration)
>>> t = np.arange(samples) / fs
```

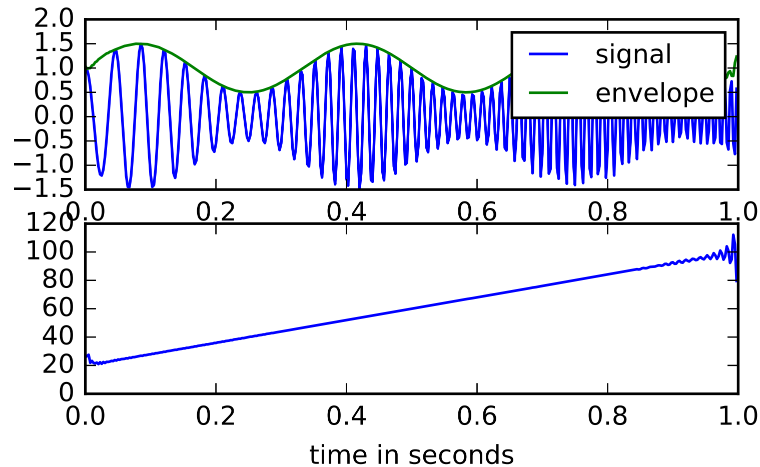
We create a chirp of which the frequency increases from 20 Hz to 100 Hz and apply an amplitude modulation.

```
>>> signal = chirp(t, 20.0, t[-1], 100.0)
>>> signal *= (1.0 + 0.5 * np.sin(2.0*np.pi*3.0*t) )
```

The amplitude envelope is given by magnitude of the analytic signal. The instantaneous frequency can be obtained by differentiating the instantaneous phase in respect to time. The instantaneous phase corresponds to the phase angle of the analytic signal.

```
>>> analytic_signal = hilbert(signal)
>>> amplitude_envelope = np.abs(analytic_signal)
>>> instantaneous_phase = np.unwrap(np.angle(analytic_signal))
>>> instantaneous_frequency = (np.diff(instantaneous_phase) /
...                            (2.0*np.pi) * fs)
```

```
>>> fig = plt.figure()
>>> ax0 = fig.add_subplot(211)
>>> ax0.plot(t, signal, label='signal')
>>> ax0.plot(t, amplitude_envelope, label='envelope')
>>> ax0.set_xlabel("time in seconds")
>>> ax0.legend()
>>> ax1 = fig.add_subplot(212)
>>> ax1.plot(t[1:], instantaneous_frequency)
>>> ax1.set_xlabel("time in seconds")
>>> ax1.set_ylim(0.0, 120.0)
```



`scipy.signal.hilbert2(x, N=None)`
Compute the '2-D' analytic signal of *x*

Parameters
x : array_like
2-D signal data.
N : int or tuple of two ints, optional
Number of Fourier components. Default is `x.shape`

Returns
xa : ndarray
Analytic signal of *x* taken along axes (0,1).

References

[R238]

`scipy.signal.decimate(x, q, n=None, ftype='iir', axis=-1, zero_phase=None)`
Downsample the signal after applying an anti-aliasing filter.

By default, an order 8 Chebyshev type I filter is used. A 30 point FIR filter with Hamming window is used if *ftype* is 'fir'.

Parameters
x : ndarray
The signal to be downsampled, as an N-dimensional array.
q : int
The downsampling factor. For downsampling factors higher than 13, it is recommended to call `decimate` multiple times.
n : int, optional
The order of the filter (1 less than the length for 'fir'). Defaults to 8 for 'iir' and 30 for 'fir'.
ftype : str {'iir', 'fir'} or `dlti` instance, optional
If 'iir' or 'fir', specifies the type of lowpass filter. If an instance of an `dlti` object, uses that object to filter before downsampling.
axis : int, optional
The axis along which to decimate.
zero_phase : bool, optional
Prevent phase shift by filtering with `filtfilt` instead of `lfilter` when using an IIR filter, and shifting the outputs back by the filter's group delay when using an FIR filter. A value of `True` is recommended, since a phase shift is generally not desired. Using `None` defaults to `False` for backwards

compatibility. This default will change to `True` in a future release, so it is best to set this argument explicitly.

New in version 0.18.0.

Returns **y** : ndarray

The down-sampled signal.

See also:

resample Resample up or down using the FFT method.

resample_poly Resample using polyphase filtering and an FIR filter.

Notes

The `zero_phase` keyword was added in 0.18.0. The possibility to use instances of `dlti` as `ftype` was added in 0.18.0.

`scipy.signal.detrend`(*data*, *axis=-1*, *type='linear'*, *bp=0*)

Remove linear trend along axis from data.

Parameters **data** : array_like

The input data.

axis : int, optional

The axis along which to detrend the data. By default this is the last axis (-1).

type : {'linear', 'constant'}, optional

The type of detrending. If `type == 'linear'` (default), the result of a linear least-squares fit to *data* is subtracted from *data*. If `type == 'constant'`, only the mean of *data* is subtracted.

bp : array_like of ints, optional

A sequence of break points. If given, an individual linear fit is performed for each part of *data* between two break points. Break points are specified as indices into *data*.

Returns **ret** : ndarray

The detrended input data.

Examples

```
>>> from scipy import signal
>>> randgen = np.random.RandomState(9)
>>> npoints = 1000
>>> noise = randgen.randn(npoints)
>>> x = 3 + 2*np.linspace(0, 1, npoints) + noise
>>> (signal.detrend(x) - noise).max() < 0.01
True
```

`scipy.signal.resample`(*x*, *num*, *t=None*, *axis=0*, *window=None*)

Resample *x* to *num* samples using Fourier method along the given axis.

The resampled signal starts at the same value as *x* but is sampled with a spacing of `len(x) / num * (spacing of x)`. Because a Fourier method is used, the signal is assumed to be periodic.

Parameters **x** : array_like

The data to be resampled.

num : int

The number of samples in the resampled signal.

t : array_like, optional

If *t* is given, it is assumed to be the sample positions associated with the signal data in *x*.

axis : int, optional
The axis of *x* that is resampled. Default is 0.

window : array_like, callable, string, float, or tuple, optional
Specifies the window applied to the signal in the Fourier domain. See below for details.

Returns resampled_x or (resampled_x, resampled_t)
Either the resampled array, or, if *t* was given, a tuple containing the resampled array and the corresponding resampled positions.

See also:

decimate Downsample the signal after applying an FIR or IIR filter.

resample_poly Resample using polyphase filtering and an FIR filter.

Notes

The argument *window* controls a Fourier-domain window that tapers the Fourier spectrum before zero-padding to alleviate ringing in the resampled values for sampled signals you didn't intend to be interpreted as band-limited.

If *window* is a function, then it is called with a vector of inputs indicating the frequency bins (i.e. fftfreq(x.shape[axis])).

If *window* is an array of the same length as *x.shape[axis]* it is assumed to be the window to be applied directly in the Fourier domain (with dc and low-frequency first).

For any other type of *window*, the function `scipy.signal.get_window` is called to generate the window.

The first sample of the returned vector is the same as the first sample of the input vector. The spacing between samples is changed from `dx` to `dx * len(x) / num`.

If *t* is not None, then it represents the old sample positions, and the new sample positions will be returned as well as the new samples.

As noted, `resample` uses FFT transformations, which can be very slow if the number of input or output samples is large and prime; see `scipy.fftpack.fft`.

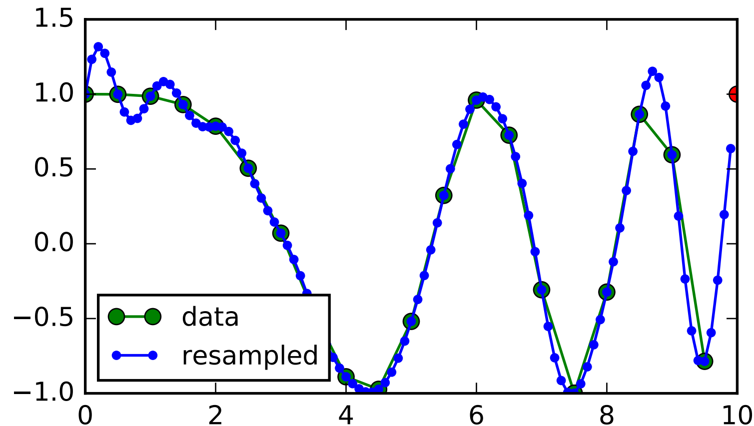
Examples

Note that the end of the resampled data rises to meet the first sample of the next cycle:

```
>>> from scipy import signal
```

```
>>> x = np.linspace(0, 10, 20, endpoint=False)
>>> y = np.cos(-x**2/6.0)
>>> f = signal.resample(y, 100)
>>> xnew = np.linspace(0, 10, 100, endpoint=False)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'go-', xnew, f, '.-', 10, y[0], 'ro')
>>> plt.legend(['data', 'resampled'], loc='best')
>>> plt.show()
```



`scipy.signal.resample_poly`(*x*, *up*, *down*, *axis=0*, *window=('kaiser', 5.0)*)

Resample *x* along the given axis using polyphase filtering.

The signal *x* is upsampled by the factor *up*, a zero-phase low-pass FIR filter is applied, and then it is downsampled by the factor *down*. The resulting sample rate is `up / down` times the original sample rate. Values beyond the boundary of the signal are assumed to be zero during the filtering step.

Parameters

x : array_like
: The data to be resampled.

up : int
: The upsampling factor.

down : int
: The downsampling factor.

axis : int, optional
: The axis of *x* that is resampled. Default is 0.

window : string, tuple, or array_like, optional
: Desired window to use to design the low-pass filter, or the FIR filter coefficients to employ. See below for details.

Returns

resampled_x : array
: The resampled array.

See also:

`decimate` Downsample the signal after applying an FIR or IIR filter.
`resample` Resample up or down using the FFT method.

Notes

This polyphase method will likely be faster than the Fourier method in *`scipy.signal.resample`* when the number of samples is large and prime, or when the number of samples is large and *up* and *down* share a large greatest common denominator. The length of the FIR filter used will depend on `max(up, down) // gcd(up, down)`, and the number of operations during polyphase filtering will depend on the filter length and *down* (see *`scipy.signal.upfirdn`* for details).

The argument *window* specifies the FIR low-pass filter design.

If *window* is an array_like it is assumed to be the FIR filter coefficients. Note that the FIR filter is applied after the upsampling step, so it should be designed to operate on a signal at a sampling frequency higher than the original by a factor of *up//gcd(up, down)*. This function's output will be centered with respect to this array, so it

is best to pass a symmetric filter with an odd number of samples if, as is usually the case, a zero-phase filter is desired.

For any other type of *window*, the functions `scipy.signal.get_window` and `scipy.signal.firwin` are called to generate the appropriate filter coefficients.

The first sample of the returned vector is the same as the first sample of the input vector. The spacing between samples is changed from `dx` to `dx * up / float(down)`.

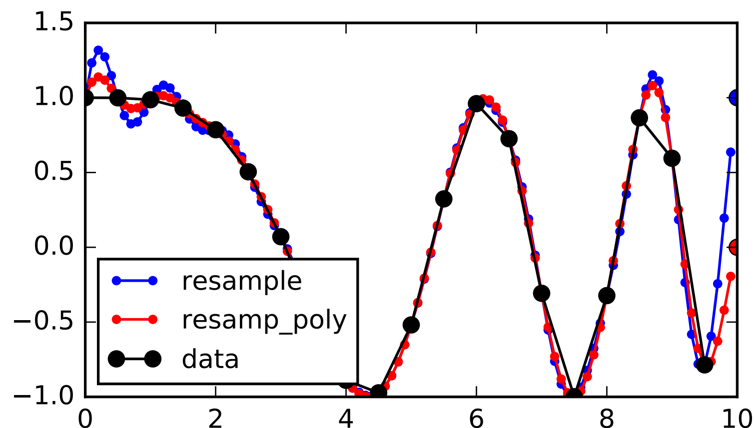
Examples

Note that the end of the resampled data rises to meet the first sample of the next cycle for the FFT method, and gets closer to zero for the polyphase method:

```
>>> from scipy import signal
```

```
>>> x = np.linspace(0, 10, 20, endpoint=False)
>>> y = np.cos(-x**2/6.0)
>>> f_fft = signal.resample(y, 100)
>>> f_poly = signal.resample_poly(y, 100, 20)
>>> xnew = np.linspace(0, 10, 100, endpoint=False)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(xnew, f_fft, 'b.-', xnew, f_poly, 'r.-')
>>> plt.plot(x, y, 'ko-')
>>> plt.plot(10, y[0], 'bo', 10, 0., 'ro')  # boundaries
>>> plt.legend(['resample', 'resamp_poly', 'data'], loc='best')
>>> plt.show()
```



`scipy.signal.`**`upfirdn`**(*h, x, up=1, down=1, axis=-1*)

Upsample, FIR filter, and downsample

Parameters

h : array_like
: 1-dimensional FIR (finite-impulse response) filter coefficients.

x : array_like
: Input signal array.

up : int, optional
: Upsampling rate. Default is 1.

down : int, optional
: Downsampling rate. Default is 1.

axis : int, optional
: The axis of the input data array along which to apply the linear filter. The filter is applied to each subarray along this axis. Default is -1.

Returns

y : ndarray
: The output signal array. Dimensions will be the same as *x* except for along *axis*, which will change size according to the *h*, *up*, and *down* parameters.

Notes

The algorithm is an implementation of the block diagram shown on page 129 of the Vaidyanathan text *[R269]* (Figure 4.3-8d).

The direct approach of upsampling by factor of P with zero insertion, FIR filtering of length `N`, and downsampling by factor of Q is O(N*Q) per output sample. The polyphase implementation used here is O(N/P).

New in version 0.18.

Examples

Simple operations:

```
>>> from scipy.signal import upfirdn
>>> upfirdn([1, 1, 1], [1, 1, 1])   # FIR filter
array([ 1.,  2.,  3.,  2.,  1.])
>>> upfirdn([1], [1, 2, 3], 3)  # upsampling with zeros insertion
array([ 1.,  0.,  0.,  2.,  0.,  0.,  3.,  0.,  0.])
>>> upfirdn([1, 1, 1], [1, 2, 3], 3)  # upsampling with sample-and-hold
array([ 1.,  1.,  1.,  2.,  2.,  2.,  3.,  3.,  3.])
>>> upfirdn([.5, 1, .5], [1, 1, 1], 2)  # linear interpolation
array([ 0.5,  1. ,  1. ,  1. ,  1. ,  1. ,  0.5,  0. ])
>>> upfirdn([1], np.arange(10), 1, 3)  # decimation by 3
array([ 0.,  3.,  6.,  9.])
>>> upfirdn([.5, 1, .5], np.arange(10), 2, 3)  # linear interp, rate 2/3
array([ 0. ,  1. ,  2.5,  4. ,  5.5,  7. ,  8.5,  0. ])
```

Apply a single filter to multiple signals:

```
>>> x = np.reshape(np.arange(8), (4, 2))
>>> x
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Apply along the last dimension of `x`:

```
>>> h = [1, 1]
>>> upfirdn(h, x, 2)
array([[ 0.,  0.,  1.,  1.],
       [ 2.,  2.,  3.,  3.],
       [ 4.,  4.,  5.,  5.],
       [ 6.,  6.,  7.,  7.]])
```

Apply along the 0th dimension of `x`:

```
>>> upfirdn(h, x, 2, axis=0)
array([[ 0.,  1.],
       [ 0.,  1.],
```

```
       [ 2.,  3.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 6.,  7.]])
```

5.20.4 Filter design

bilinear(b, a[, fs])	Return a digital filter from an analog one using a bilinear transform.
findfreqs(num, den, N[, kind])	Find array of frequencies for computing the response of an analog filter.
firls(numtaps, bands, desired[, weight, nyq])	FIR filter design using least-squares error minimization.
firwin(numtaps, cutoff[, width, window, ...])	FIR filter design using the window method.
firwin2(numtaps, freq, gain[, nfreqs, ...])	FIR filter design using the window method.
freqs(b, a[, worN, plot])	Compute frequency response of analog filter.
freqs_zpk(z, p, k[, worN])	Compute frequency response of analog filter.
freqz(b[, a, worN, whole, plot])	Compute the frequency response of a digital filter.
freqz_zpk(z, p, k[, worN, whole])	Compute the frequency response of a digital filter in ZPK form.
sosfreqz(sos[, worN, whole])	Compute the frequency response of a digital filter in SOS format.
group_delay(system[, w, whole])	Compute the group delay of a digital filter.
iirdesign(wp, ws, gpass, gstop[, analog, ...])	Complete IIR digital and analog filter design.
iirfilter(N, Wn[, rp, rs, btype, analog, ...])	IIR digital and analog filter design given order and critical points.
kaiser_atten(numtaps, width)	Compute the attenuation of a Kaiser FIR filter.
kaiser_beta(a)	Compute the Kaiser parameter *beta*, given the attenuation *a*.
kaiserord(ripple, width)	Design a Kaiser window to limit ripple and width of transition region.
minimum_phase(h[, method, n_fft])	Convert a linear-phase FIR filter to minimum phase
savgol_coeffs(window_length, polyorder[, ...])	Compute the coefficients for a 1-d Savitzky-Golay FIR filter.
remez(numtaps, bands, desired[, weight, Hz, ...])	Calculate the minimax optimal filter using the Remez exchange algorithm.
unique_roots(p[, tol, rtype])	Determine unique roots and their multiplicities from a list of roots.
residue(b, a[, tol, rtype])	Compute partial-fraction expansion of b(s) / a(s).
residuez(b, a[, tol, rtype])	Compute partial-fraction expansion of b(z) / a(z).
invres(r, p, k[, tol, rtype])	Compute b(s) and a(s) from partial fraction expansion.
invresz(r, p, k[, tol, rtype])	Compute b(z) and a(z) from partial fraction expansion.
BadCoefficients	Warning about badly conditioned filter coefficients

scipy.signal.**bilinear**(*b*, *a*, *fs=1.0*)

Return a digital filter from an analog one using a bilinear transform.

The bilinear transform substitutes `(z-1) / (z+1)` for `s`.

scipy.signal.**findfreqs**(*num*, *den*, *N*, *kind='ba'*)

Find array of frequencies for computing the response of an analog filter.

Parameters
num, den : array_like, 1-D
The polynomial coefficients of the numerator and denominator of the transfer function of the filter or LTI system, where the coefficients are ordered from highest to lowest degree. Or, the roots of the transfer function numerator and denominator (i.e. zeroes and poles).
N : int
The length of the array to be computed.
kind : str {'ba', 'zp'}, optional
Specifies whether the numerator and denominator are specified by their polynomial coefficients ('ba'), or their roots ('zp').

Returns
w : (N,) ndarray
A 1-D array of frequencies, logarithmically spaced.

Examples

Find a set of nine frequencies that span the "interesting part" of the frequency response for the filter with the transfer function

H(s) = s / (s^2 + 8s + 25)

```
>>> from scipy import signal
>>> signal.findfreqs([1, 0], [1, 8, 25], N=9)
array([  1.00000000e-02,   3.16227766e-02,   1.00000000e-01,
         3.16227766e-01,   1.00000000e+00,   3.16227766e+00,
         1.00000000e+01,   3.16227766e+01,   1.00000000e+02])
```

`scipy.signal.`**`firls`**(*numtaps*, *bands*, *desired*, *weight=None*, *nyq=1.0*)
FIR filter design using least-squares error minimization.

Calculate the filter coefficients for the linear-phase finite impulse response (FIR) filter which has the best approximation to the desired frequency response described by *bands* and *desired* in the least squares sense (i.e., the integral of the weighted mean-squared error within the specified bands is minimized).

Parameters
numtaps : int
The number of taps in the FIR filter. *numtaps* must be odd.
bands : array_like
A monotonic nondecreasing sequence containing the band edges in Hz. All elements must be non-negative and less than or equal to the Nyquist frequency given by *nyq*.
desired : array_like
A sequence the same size as *bands* containing the desired gain at the start and end point of each band.
weight : array_like, optional
A relative weighting to give to each band region when solving the least squares problem. *weight* has to be half the size of *bands*.
nyq : float, optional
Nyquist frequency. Each frequency in *bands* must be between 0 and *nyq* (inclusive).

Returns
coeffs : ndarray
Coefficients of the optimal (in a least squares sense) FIR filter.

See also:

firwin, *firwin2*, *minimum_phase*, *remez*

Notes

This implementation follows the algorithm given in *[R218]*. As noted there, least squares design has multiple advantages:

1. Optimal in a least-squares sense.
2. Simple, non-iterative method.
3. The general solution can obtained by solving a linear system of equations.
4. Allows the use of a frequency dependent weighting function.

This function constructs a Type I linear phase FIR filter, which contains an odd number of *coeffs* satisfying for $n < numtaps$:

$$coeffs(n) = coeffs(numtaps - 1 - n)$$

The odd number of coefficients and filter symmetry avoid boundary conditions that could otherwise occur at the Nyquist and 0 frequencies (e.g., for Type II, III, or IV variants).

New in version 0.18.

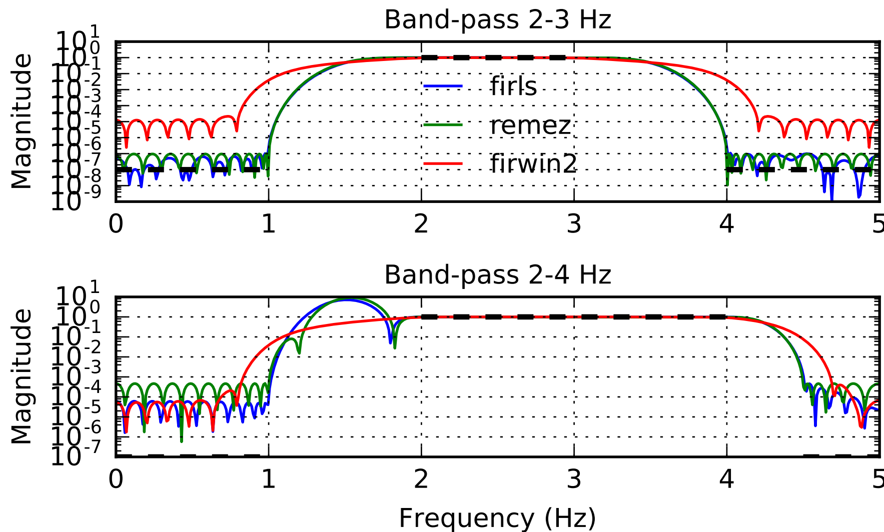
References

[R218]

Examples

We want to construct a band-pass filter. Note that the behavior in the frequency ranges between our stop bands and pass bands is unspecified, and thus may overshoot depending on the parameters of our filter:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> fig, axs = plt.subplots(2)
>>> nyq = 5.  # Hz
>>> desired = (0, 0, 1, 1, 0, 0)
>>> for bi, bands in enumerate(((0, 1, 2, 3, 4, 5), (0, 1, 2, 4, 4.5, 5))):
...     fir_firls = signal.firls(73, bands, desired, nyq=nyq)
...     fir_remez = signal.remez(73, bands, desired[::2], Hz=2 * nyq)
...     fir_firwin2 = signal.firwin2(73, bands, desired, nyq=nyq)
...     hs = list()
...     ax = axs[bi]
...     for fir in (fir_firls, fir_remez, fir_firwin2):
...         freq, response = signal.freqz(fir)
...         hs.append(ax.semilogy(nyq*freq/(np.pi), np.abs(response))[0])
...     for band, gains in zip(zip(bands[::2], bands[1::2]), zip(desired[::2],
↪desired[1::2])):
...         ax.semilogy(band, np.maximum(gains, 1e-7), 'k--', linewidth=2)
...     if bi == 0:
...         ax.legend(hs, ('firls', 'remez', 'firwin2'), loc='lower center',
↪frameon=False)
...     else:
...         ax.set_xlabel('Frequency (Hz)')
...     ax.grid(True)
...     ax.set(title='Band-pass %d-%d Hz' % bands[2:4], ylabel='Magnitude')
...
>>> fig.tight_layout()
>>> plt.show()
```



`scipy.signal.firwin`(*numtaps*, *cutoff*, *width=None*, *window='hamming'*, *pass_zero=True*, *scale=True*, *nyq=1.0*)

FIR filter design using the window method.

This function computes the coefficients of a finite impulse response filter. The filter will have linear phase; it will be Type I if *numtaps* is odd and Type II if *numtaps* is even.

Type II filters always have zero response at the Nyquist rate, so a ValueError exception is raised if firwin is called with *numtaps* even and having a passband whose right end is at the Nyquist rate.

Parameters

numtaps : int
: Length of the filter (number of coefficients, i.e. the filter order + 1). *numtaps* must be even if a passband includes the Nyquist frequency.

cutoff : float or 1D array_like
: Cutoff frequency of filter (expressed in the same units as *nyq*) OR an array of cutoff frequencies (that is, band edges). In the latter case, the frequencies in *cutoff* should be positive and monotonically increasing between 0 and *nyq*. The values 0 and *nyq* must not be included in *cutoff*.

width : float or None, optional
: If *width* is not None, then assume it is the approximate width of the transition region (expressed in the same units as *nyq*) for use in Kaiser FIR filter design. In this case, the *window* argument is ignored.

window : string or tuple of string and parameter values, optional
: Desired window to use. See `scipy.signal.get_window` for a list of windows and required parameters.

pass_zero : bool, optional
: If True, the gain at the frequency 0 (i.e. the "DC gain") is 1. Otherwise the DC gain is 0.

scale : bool, optional
: Set to True to scale the coefficients so that the frequency response is exactly unity at a certain frequency. That frequency is either:
  - 0 (DC) if the first passband starts at 0 (i.e. pass_zero is True)
  - *nyq* (the Nyquist rate) if the first passband ends at *nyq* (i.e the filter is a single band highpass filter); center of first passband otherwise

nyq : float, optional
: Nyquist frequency. Each frequency in *cutoff* must be between 0 and *nyq*.

Returns

h : (numtaps,) ndarray

Coefficients of length *numtaps* FIR filter.

Raises **ValueError**

If any value in *cutoff* is less than or equal to 0 or greater than or equal to *nyq*, if the values in *cutoff* are not strictly monotonically increasing, or if *numtaps* is even but a passband includes the Nyquist frequency.

See also:

`firwin2`, `firls`, `minimum_phase`, `remez`

Examples

Low-pass from 0 to f:

```
>>> from scipy import signal
>>> numtaps = 3
>>> f = 0.1
>>> signal.firwin(numtaps, f)
array([ 0.06799017,  0.86401967,  0.06799017])
```

Use a specific window function:

```
>>> signal.firwin(numtaps, f, window='nuttall')
array([  3.56607041e-04,   9.99286786e-01,   3.56607041e-04])
```

High-pass ('stop' from 0 to f):

```
>>> signal.firwin(numtaps, f, pass_zero=False)
array([-0.00859313,  0.98281375, -0.00859313])
```

Band-pass:

```
>>> f1, f2 = 0.1, 0.2
>>> signal.firwin(numtaps, [f1, f2], pass_zero=False)
array([ 0.06301614,  0.88770441,  0.06301614])
```

Band-stop:

```
>>> signal.firwin(numtaps, [f1, f2])
array([-0.00801395,  1.0160279 , -0.00801395])
```

Multi-band (passbands are [0, f1], [f2, f3] and [f4, 1]):

```
>>> f3, f4 = 0.3, 0.4
>>> signal.firwin(numtaps, [f1, f2, f3, f4])
array([-0.01376344,  1.02752689, -0.01376344])
```

Multi-band (passbands are [f1, f2] and [f3,f4]):

```
>>> signal.firwin(numtaps, [f1, f2, f3, f4], pass_zero=False)
array([ 0.04890915,  0.91284326,  0.04890915])
```

`scipy.signal.`**`firwin2`**(*numtaps*, *freq*, *gain*, *nfreqs=None*, *window='hamming'*, *nyq=1.0*, *antisymmetric=False*)

FIR filter design using the window method.

From the given frequencies *freq* and corresponding gains *gain*, this function constructs an FIR filter with linear phase and (approximately) the given frequency response.

Parameters **numtaps** : int

The number of taps in the FIR filter. *numtaps* must be less than *nfreqs*.

freq : array_like, 1D
The frequency sampling points. Typically 0.0 to 1.0 with 1.0 being Nyquist. The Nyquist frequency can be redefined with the argument *nyq*. The values in *freq* must be nondecreasing. A value can be repeated once to implement a discontinuity. The first value in *freq* must be 0, and the last value must be *nyq*.

gain : array_like
The filter gains at the frequency sampling points. Certain constraints to gain values, depending on the filter type, are applied, see Notes for details.

nfreqs : int, optional
The size of the interpolation mesh used to construct the filter. For most efficient behavior, this should be a power of 2 plus 1 (e.g, 129, 257, etc). The default is one more than the smallest power of 2 that is not less than *numtaps*. *nfreqs* must be greater than *numtaps*.

window : string or (string, float) or float, or None, optional
Window function to use. Default is "hamming". See `scipy.signal.get_window` for the complete list of possible values. If None, no window function is applied.

nyq : float, optional
Nyquist frequency. Each frequency in *freq* must be between 0 and *nyq* (inclusive).

antisymmetric : bool, optional
Whether resulting impulse response is symmetric/antisymmetric. See Notes for more details.

Returns

taps : ndarray
The filter coefficients of the FIR filter, as a 1-D array of length *numtaps*.

See also:

`firls`, `firwin`, `minimum_phase`, `remez`

Notes

From the given set of frequencies and gains, the desired response is constructed in the frequency domain. The inverse FFT is applied to the desired response to create the associated convolution kernel, and the first *numtaps* coefficients of this kernel, scaled by *window*, are returned.

The FIR filter will have linear phase. The type of filter is determined by the value of 'numtaps' and *antisymmetric* flag. There are four possible combinations:

- odd *numtaps*, *antisymmetric* is False, type I filter is produced
- even *numtaps*, *antisymmetric* is False, type II filter is produced
- odd *numtaps*, *antisymmetric* is True, type III filter is produced
- even *numtaps*, *antisymmetric* is True, type IV filter is produced

Magnitude response of all but type I filters are subjects to following constraints:

- type II – zero at the Nyquist frequency
- type III – zero at zero and Nyquist frequencies
- type IV – zero at zero frequency

New in version 0.9.0.

References

[R219], *[R220]*

Examples

A lowpass FIR filter with a response that is 1 on [0.0, 0.5], and that decreases linearly on [0.5, 1.0] from 1 to 0:

```
>>> from scipy import signal
>>> taps = signal.firwin2(150, [0.0, 0.5, 1.0], [1.0, 1.0, 0.0])
>>> print(taps[72:78])
[-0.02286961 -0.06362756  0.57310236  0.57310236 -0.06362756 -0.02286961]
```

`scipy.signal.freqs`(*b*, *a*, *worN=None*, *plot=None*)

Compute frequency response of analog filter.

Given the M-order numerator *b* and N-order denominator *a* of an analog filter, compute its frequency response:

```
        b[0]*(jw)**M + b[1]*(jw)**(M-1) + ... + b[M]
H(w) = ----------------------------------------------
        a[0]*(jw)**N + a[1]*(jw)**(N-1) + ... + a[N]
```

Parameters

b : array_like
: Numerator of a linear filter.

a : array_like
: Denominator of a linear filter.

worN : {None, int, array_like}, optional
: If None, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, then compute at that many frequencies. Otherwise, compute the response at the angular frequencies (e.g. rad/s) given in *worN*.

plot : callable, optional
: A callable that takes two arguments. If given, the return parameters *w* and *h* are passed to plot. Useful for plotting the frequency response inside `freqs`.

Returns

w : ndarray
: The angular frequencies at which *h* was computed.

h : ndarray
: The frequency response.

See also:

`freqz` Compute the frequency response of a digital filter.

Notes

Using Matplotlib's "plot" function as the callable for *plot* produces unexpected results, this plots the real part of the complex transfer function, not the magnitude. Try `lambda w, h: plot(w, abs(h))`.

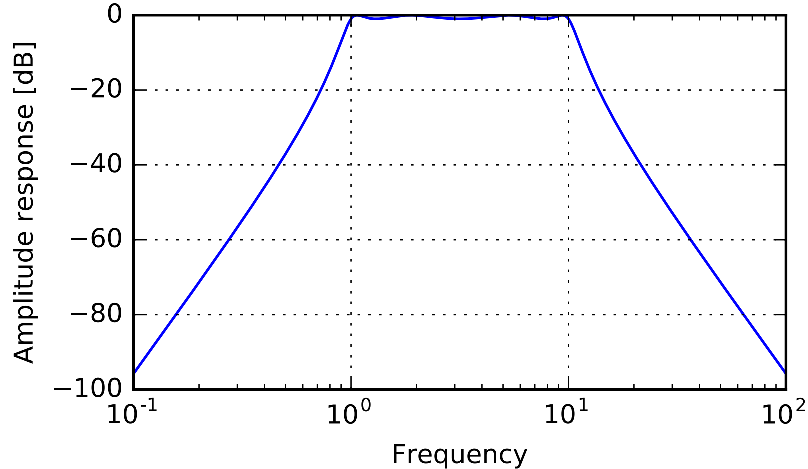
Examples

```
>>> from scipy.signal import freqs, iirfilter
```

```
>>> b, a = iirfilter(4, [1, 10], 1, 60, analog=True, ftype='cheby1')
```

```
>>> w, h = freqs(b, a, worN=np.logspace(-1, 2, 1000))
```

```
>>> import matplotlib.pyplot as plt
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.xlabel('Frequency')
>>> plt.ylabel('Amplitude response [dB]')
>>> plt.grid()
>>> plt.show()
```



`scipy.signal.`**`freqs_zpk`**(*z*, *p*, *k*, *worN=None*)

Compute frequency response of analog filter.

Given the zeros *z*, poles *p*, and gain *k* of a filter, compute its frequency response:

```
            (jw-z[0]) * (jw-z[1]) * ... * (jw-z[-1])
H(w) = k * ----------------------------------------
            (jw-p[0]) * (jw-p[1]) * ... * (jw-p[-1])
```

Parameters

z : array_like
: Zeroes of a linear filter

p : array_like
: Poles of a linear filter

k : scalar
: Gain of a linear filter

worN : {None, int, array_like}, optional
: If None, then compute at 200 frequencies around the interesting parts of the response curve (determined by pole-zero locations). If a single integer, then compute at that many frequencies. Otherwise, compute the response at the angular frequencies (e.g. rad/s) given in *worN*.

Returns

w : ndarray
: The angular frequencies at which *h* was computed.

h : ndarray
: The frequency response.

See also:

`freqs` Compute the frequency response of an analog filter in TF form
`freqz` Compute the frequency response of a digital filter in TF form
`freqz_zpk` Compute the frequency response of a digital filter in ZPK form

Notes

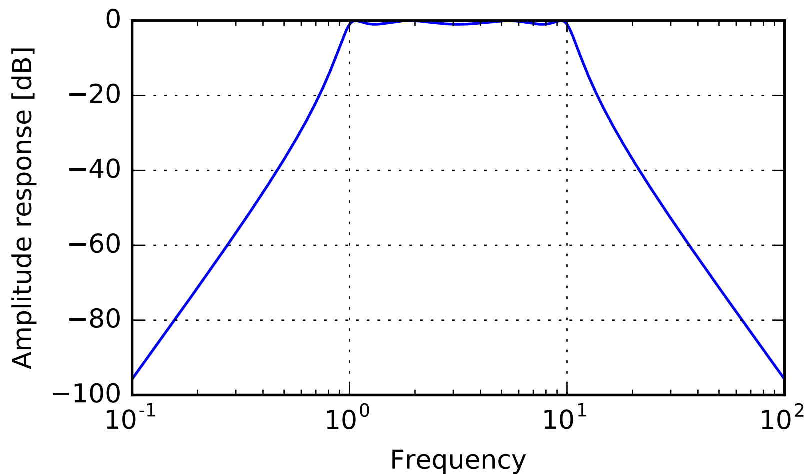
Examples

```
>>> from scipy.signal import freqs_zpk, iirfilter
```

```
>>> z, p, k = iirfilter(4, [1, 10], 1, 60, analog=True, ftype='cheby1',
...                     output='zpk')
```

```
>>> w, h = freqs_zpk(z, p, k, worN=np.logspace(-1, 2, 1000))
```

```
>>> import matplotlib.pyplot as plt
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.xlabel('Frequency')
>>> plt.ylabel('Amplitude response [dB]')
>>> plt.grid()
>>> plt.show()
```



`scipy.signal.`**`freqz`**(*b*, *a=1*, *worN=None*, *whole=False*, *plot=None*)

Compute the frequency response of a digital filter.

Given the M-order numerator *b* and N-order denominator *a* of a digital filter, compute its frequency response:

```
             jw                 -jw              -jwM
    jw    B(e  )  b[0] + b[1]e    + .... + b[M]e
 H(e  ) = ---- = -----------------------------------
             jw                 -jw              -jwN
          A(e  )  a[0] + a[1]e    + .... + a[N]e
```

Parameters **b** : array_like

numerator of a linear filter

a : array_like

denominator of a linear filter

worN : {None, int, array_like}, optional

If None (default), then compute at 512 frequencies equally spaced around the unit circle. If a single integer, then compute at that many frequencies. If an array_like, compute the response at the frequencies given (in radians/sample).

whole : bool, optional

Normally, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to 2*pi radians/sample.

plot : callable

A callable that takes two arguments. If given, the return parameters *w* and *h* are passed to plot. Useful for plotting the frequency response inside `freqz`.

Returns

w : ndarray

The normalized frequencies at which *h* was computed, in radians/sample.

h : ndarray

The frequency response, as complex numbers.

See also:

`sosfreqz`

Notes

Using Matplotlib's "plot" function as the callable for *plot* produces unexpected results, this plots the real part of the complex transfer function, not the magnitude. Try `lambda w, h: plot(w, abs(h))`.

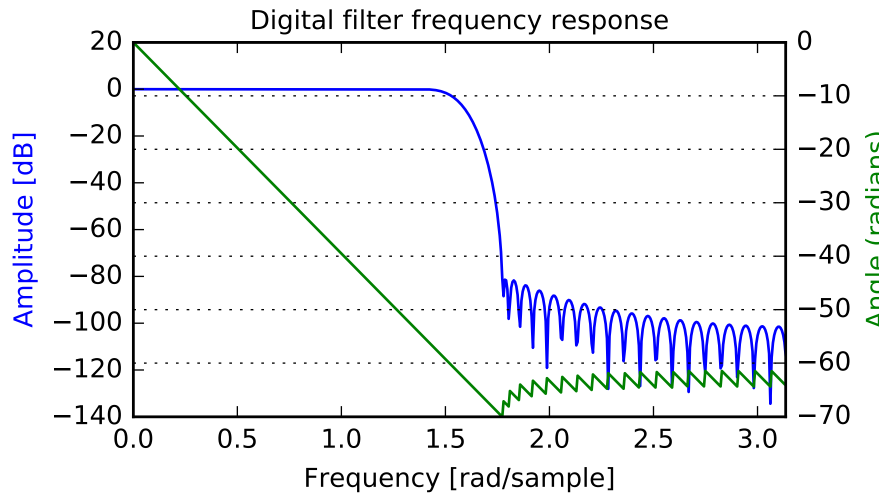
Examples

```
>>> from scipy import signal
>>> b = signal.firwin(80, 0.5, window=('kaiser', 8))
>>> w, h = signal.freqz(b)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.title('Digital filter frequency response')
>>> ax1 = fig.add_subplot(111)
```

```
>>> plt.plot(w, 20 * np.log10(abs(h)), 'b')
>>> plt.ylabel('Amplitude [dB]', color='b')
>>> plt.xlabel('Frequency [rad/sample]')
```

```
>>> ax2 = ax1.twinx()
>>> angles = np.unwrap(np.angle(h))
>>> plt.plot(w, angles, 'g')
>>> plt.ylabel('Angle (radians)', color='g')
>>> plt.grid()
>>> plt.axis('tight')
>>> plt.show()
```



scipy.signal.**freqz_zpk**(*z*, *p*, *k*, *worN=None*, *whole=False*)

Compute the frequency response of a digital filter in ZPK form.

Given the Zeros, Poles and Gain of a digital filter, compute its frequency response:

```
:math:`H(z)=k \prod_i (z - Z[i]) / \prod_j (z - P[j])`
```

where k is the *gain*, Z are the *zeros* and P are the *poles*.

Parameters

z : array_like
: Zeroes of a linear filter

p : array_like
: Poles of a linear filter

k : scalar
: Gain of a linear filter

worN : {None, int, array_like}, optional
: If None (default), then compute at 512 frequencies equally spaced around the unit circle. If a single integer, then compute at that many frequencies. If an array_like, compute the response at the frequencies given (in radians/sample).

whole : bool, optional
: Normally, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to 2*pi radians/sample.

Returns

w : ndarray
: The normalized frequencies at which *h* was computed, in radians/sample.

h : ndarray
: The frequency response.

See also:

freqs	Compute the frequency response of an analog filter in TF form
freqs_zpk	Compute the frequency response of an analog filter in ZPK form
freqz	Compute the frequency response of a digital filter in TF form

Notes

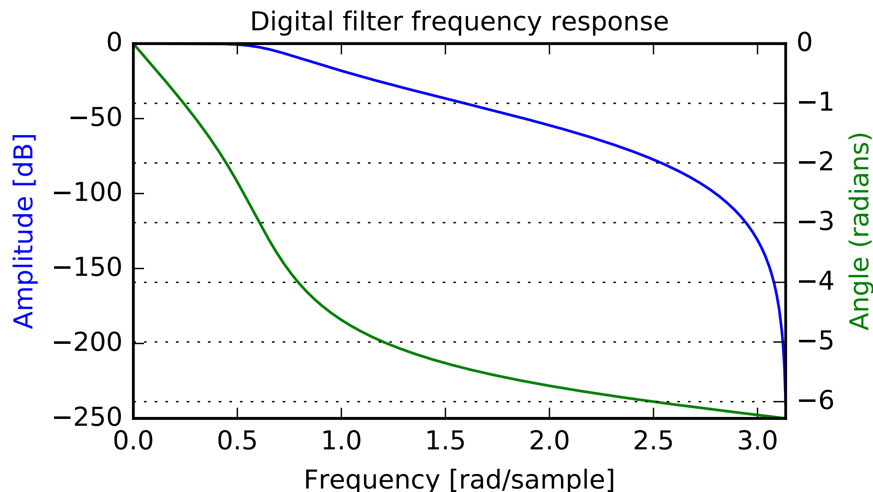
Examples

```
>>> from scipy import signal
>>> z, p, k = signal.butter(4, 0.2, output='zpk')
>>> w, h = signal.freqz_zpk(z, p, k)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.title('Digital filter frequency response')
>>> ax1 = fig.add_subplot(111)
```

```
>>> plt.plot(w, 20 * np.log10(abs(h)), 'b')
>>> plt.ylabel('Amplitude [dB]', color='b')
>>> plt.xlabel('Frequency [rad/sample]')
```

```
>>> ax2 = ax1.twinx()
>>> angles = np.unwrap(np.angle(h))
>>> plt.plot(w, angles, 'g')
>>> plt.ylabel('Angle (radians)', color='g')
>>> plt.grid()
>>> plt.axis('tight')
>>> plt.show()
```



`scipy.signal.`**`sosfreqz`**(*sos*, *worN=None*, *whole=False*)

Compute the frequency response of a digital filter in SOS format.

Given *sos*, an array with shape (n, 6) of second order sections of a digital filter, compute the frequency response of the system function:

```
       B0(z)    B1(z)          B{n-1}(z)
H(z) = ----- * ----- * ... * ---------
       A0(z)    A1(z)          A{n-1}(z)
```

for z = exp(omega*1j), where B{k}(z) and A{k}(z) are numerator and denominator of the transfer function of the k-th second order section.

Parameters

sos : array_like
Array of second-order filter coefficients, must have shape `(n_sections, 6)`. Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.

worN : {None, int, array_like}, optional
If None (default), then compute at 512 frequencies equally spaced around the unit circle. If a single integer, then compute at that many frequencies. If an array_like, compute the response at the frequencies given (in radians/sample).

whole : bool, optional
Normally, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to 2*pi radians/sample.

Returns

w : ndarray
The normalized frequencies at which *h* was computed, in radians/sample.

h : ndarray
The frequency response, as complex numbers.

See also:

freqz, *sosfilt*

Notes

New in version 0.19.0.

Examples

Design a 15th-order bandpass filter in SOS format.

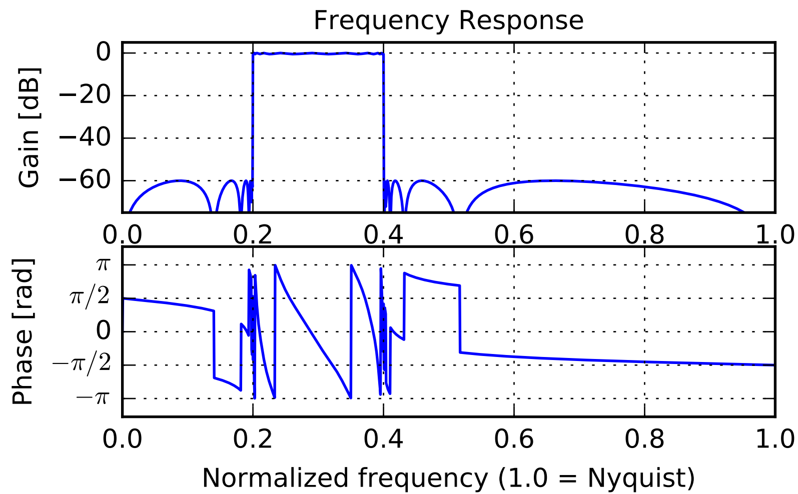
```
>>> from scipy import signal
>>> sos = signal.ellip(15, 0.5, 60, (0.2, 0.4), btype='bandpass',
...                    output='sos')
```

Compute the frequency response at 1500 points from DC to Nyquist.

```
>>> w, h = signal.sosfreqz(sos, worN=1500)
```

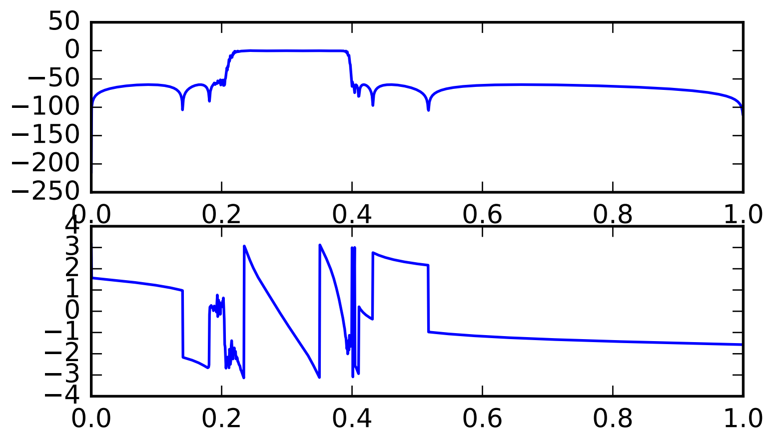
Plot the response.

```
>>> import matplotlib.pyplot as plt
>>> plt.subplot(2, 1, 1)
>>> db = 20*np.log10(np.abs(h))
>>> plt.plot(w/np.pi, db)
>>> plt.ylim(-75, 5)
>>> plt.grid(True)
>>> plt.yticks([0, -20, -40, -60])
>>> plt.ylabel('Gain [dB]')
>>> plt.title('Frequency Response')
>>> plt.subplot(2, 1, 2)
>>> plt.plot(w/np.pi, np.angle(h))
>>> plt.grid(True)
>>> plt.yticks([-np.pi, -0.5*np.pi, 0, 0.5*np.pi, np.pi],
...            [r'$-\pi$', r'$-\pi/2$', '0', r'$\pi/2$', r'$\pi$'])
>>> plt.ylabel('Phase [rad]')
>>> plt.xlabel('Normalized frequency (1.0 = Nyquist)')
>>> plt.show()
```



If the same filter is implemented as a single transfer function, numerical error corrupts the frequency response:

```
>>> b, a = signal.ellip(15, 0.5, 60, (0.2, 0.4), btype='bandpass',
...                     output='ba')
>>> w, h = signal.freqz(b, a, worN=1500)
>>> plt.subplot(2, 1, 1)
>>> db = 20*np.log10(np.abs(h))
>>> plt.plot(w/np.pi, db)
>>> plt.subplot(2, 1, 2)
>>> plt.plot(w/np.pi, np.angle(h))
>>> plt.show()
```



`scipy.signal.`**`group_delay`**(*system*, *w=None*, *whole=False*)

Compute the group delay of a digital filter.

The group delay measures by how many samples amplitude envelopes of various spectral components of a signal are delayed by a filter. It is formally defined as the derivative of continuous (unwrapped) phase:

```
           d        jw
D(w) = - -- arg H(e)
          dw
```

Parameters

system : tuple of array_like (b, a)
: Numerator and denominator coefficients of a filter transfer function.

w : {None, int, array-like}, optional
: If None (default), then compute at 512 frequencies equally spaced around the unit circle. If a single integer, then compute at that many frequencies. If array, compute the delay at the frequencies given (in radians/sample).

whole : bool, optional
: Normally, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to `2*pi` radians/sample.

Returns

w : ndarray
: The normalized frequencies at which the group delay was computed, in radians/sample.

gd : ndarray
: The group delay.

See also:

freqz Frequency response of a digital filter

Notes

The similar function in MATLAB is called *grpdelay*.

If the transfer function $H(z)$ has zeros or poles on the unit circle, the group delay at corresponding frequencies is undefined. When such a case arises the warning is raised and the group delay is set to 0 at those frequencies.

For the details of numerical computation of the group delay refer to *[R222]*.

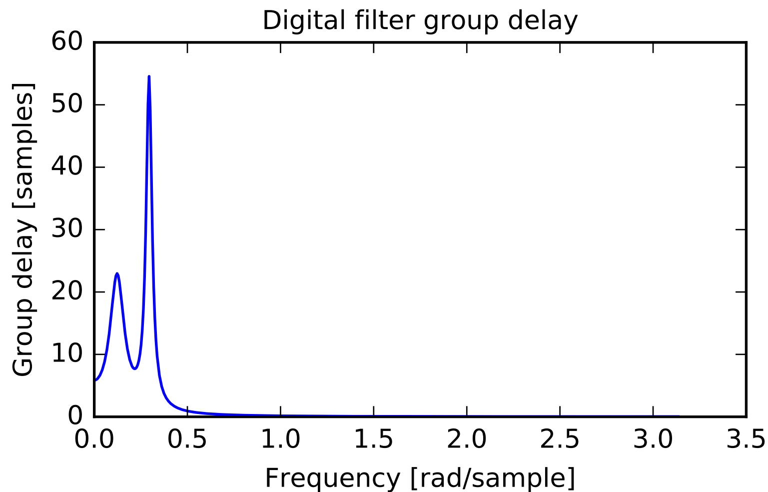
References

[R222]

Examples

```
>>> from scipy import signal
>>> b, a = signal.iirdesign(0.1, 0.3, 5, 50, ftype='cheby1')
>>> w, gd = signal.group_delay((b, a))
```

```
>>> import matplotlib.pyplot as plt
>>> plt.title('Digital filter group delay')
>>> plt.plot(w, gd)
>>> plt.ylabel('Group delay [samples]')
>>> plt.xlabel('Frequency [rad/sample]')
>>> plt.show()
```



`scipy.signal.iirdesign`(*wp*, *ws*, *gpass*, *gstop*, *analog=False*, *ftype='ellip'*, *output='ba'*)

Complete IIR digital and analog filter design.

Given passband and stopband frequencies and gains, construct an analog or digital IIR filter of minimum order for a given basic type. Return the output in numerator, denominator ('ba'), pole-zero ('zpk') or second order sections ('sos') form.

Parameters

wp, ws : float

Passband and stopband edge frequencies. For digital filters, these are normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: wp = 0.2, ws = 0.3
- Highpass: wp = 0.3, ws = 0.2
- Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6]
- Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g. rad/s).

gpass : float

The maximum loss in the passband (dB).

gstop : float

The minimum attenuation in the stopband (dB).

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

ftype : str, optional

The type of IIR filter to design:

- Butterworth : 'butter'
- Chebyshev I : 'cheby1'
- Chebyshev II : 'cheby2'
- Cauer/elliptic: 'ellip'
- Bessel/Thomson: 'bessel'

output : {'ba', 'zpk', 'sos'}, optional

Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns

b, a : ndarray, ndarray

Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float

Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray

Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

butter Filter design using order and critical points

cheby1, *cheby2*, *ellip*, *bessel*

buttord Find order and critical points from passband and stopband spec

cheb1ord, *cheb2ord*, *ellipord*

iirfilter General filter design using order and critical frequencies

Notes

The `'sos'` output parameter was added in 0.16.0.

`scipy.signal.`**iirfilter**(*N*, *Wn*, *rp=None*, *rs=None*, *btype='band'*, *analog=False*, *ftype='butter'*, *output='ba'*)

IIR digital and analog filter design given order and critical points.

Design an Nth-order digital or analog filter and return the filter coefficients.

Parameters

N : int

The order of the filter.

Wn : array_like

A scalar or length-2 sequence giving the critical frequencies. For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.) For analog filters, *Wn* is an angular frequency (e.g. rad/s).

rp : float, optional

For Chebyshev and elliptic filters, provides the maximum ripple in the passband. (dB)

rs : float, optional

For Chebyshev and elliptic filters, provides the minimum attenuation in the stop band. (dB)

btype : {'bandpass', 'lowpass', 'highpass', 'bandstop'}, optional

The type of filter. Default is 'bandpass'.

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

ftype : str, optional

The type of IIR filter to design:

- Butterworth : 'butter'
- Chebyshev I : 'cheby1'
- Chebyshev II : 'cheby2'
- Cauer/elliptic: 'ellip'
- Bessel/Thomson: 'bessel'

output : {'ba', 'zpk', 'sos'}, optional

Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns

b, a : ndarray, ndarray

Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float

Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray

Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

butter Filter design using order and critical points

cheby1, *cheby2*, *ellip*, *bessel*

buttord Find order and critical points from passband and stopband spec

cheb1ord, *cheb2ord*, *ellipord*

iirdesign General filter design using passband and stopband spec

Notes

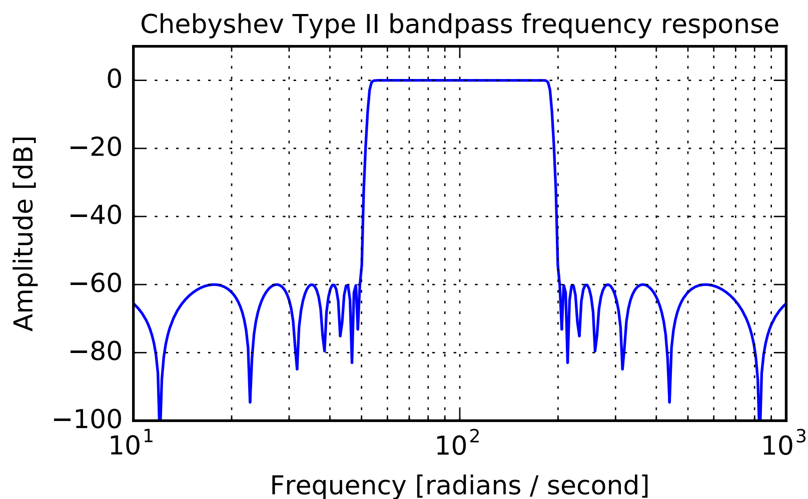
The `'sos'` output parameter was added in 0.16.0.

Examples

Generate a 17th-order Chebyshev II bandpass filter and plot the frequency response:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.iirfilter(17, [50, 200], rs=60, btype='band',
...                         analog=True, ftype='cheby2')
>>> w, h = signal.freqs(b, a, 1000)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.semilogx(w, 20 * np.log10(abs(h)))
>>> ax.set_title('Chebyshev Type II bandpass frequency response')
>>> ax.set_xlabel('Frequency [radians / second]')
>>> ax.set_ylabel('Amplitude [dB]')
>>> ax.axis((10, 1000, -100, 10))
>>> ax.grid(which='both', axis='both')
>>> plt.show()
```



`scipy.signal.kaiser_atten(numtaps, width)`

Compute the attenuation of a Kaiser FIR filter.

Given the number of taps *N* and the transition width *width*, compute the attenuation *a* in dB, given by Kaiser's formula:

a = 2.285 * (N - 1) * pi * width + 7.95

Parameters **numtaps** : int
The number of taps in the FIR filter.
width : float
The desired width of the transition region between passband and stopband (or, in general, at any discontinuity) for the filter.

Returns **a** : float
The attenuation of the ripple, in dB.

See also:

kaiserord, *kaiser_beta*

`scipy.signal.`**`kaiser_beta`**`(`*a*`)`
Compute the Kaiser parameter *beta*, given the attenuation *a*.

Parameters **a** : float
The desired attenuation in the stopband and maximum ripple in the passband, in dB. This should be a *positive* number.

Returns **beta** : float
The *beta* parameter to be used in the formula for a Kaiser window.

References

Oppenheim, Schafer, "Discrete-Time Signal Processing", p.475-476.

`scipy.signal.`**`kaiserord`**`(`*ripple*, *width*`)`
Design a Kaiser window to limit ripple and width of transition region.

Parameters **ripple** : float
Positive number specifying maximum ripple in passband (dB) and minimum ripple in stopband.
width : float
Width of transition region (normalized so that 1 corresponds to pi radians / sample).

Returns **numtaps** : int
The length of the kaiser window.
beta : float
The beta parameter for the kaiser window.

See also:

kaiser_beta, *kaiser_atten*

Notes

There are several ways to obtain the Kaiser window:
- `signal.kaiser(numtaps, beta, sym=True)`
- `signal.get_window(beta, numtaps)`
- `signal.get_window(('kaiser', beta), numtaps)`

The empirical equations discovered by Kaiser are used.

References

Oppenheim, Schafer, "Discrete-Time Signal Processing", p.475-476.

`scipy.signal.`**`minimum_phase`**`(`*h*, *method='homomorphic'*, *n_fft=None*`)`
Convert a linear-phase FIR filter to minimum phase

Parameters **h** : array
Linear-phase FIR filter coefficients.
method : {'hilbert', 'homomorphic'}
The method to use:

'homomorphic' (default) This method *[R253]* *[R254]* works best with filters with an odd number of taps, and the resulting minimum phase filter will have a magnitude response that approximates the square root of the the original filter's magnitude response.

'hilbert' This method *[R250]* is designed to be used with equiripple filters (e.g., from `remez`) with unity or zero gain regions.

n_fft : int
The number of points to use for the FFT. Should be at least a few times larger than the signal length (see Notes).

Returns **h_minimum** : array
The minimum-phase version of the filter, with length `(length(h) + 1) // 2`.

See also:

`firwin`, `firwin2`, `remez`

Notes

Both the Hilbert *[R250]* or homomorphic *[R253]* *[R254]* methods require selection of an FFT length to estimate the complex cepstrum of the filter.

In the case of the Hilbert method, the deviation from the ideal spectrum `epsilon` is related to the number of stopband zeros `n_stop` and FFT length `n_fft` as:

```
epsilon = 2. * n_stop / n_fft
```

For example, with 100 stopband zeros and a FFT length of 2048, `epsilon = 0.0976`. If we conservatively assume that the number of stopband zeros is one less than the filter length, we can take the FFT length to be the next power of 2 that satisfies `epsilon=0.01` as:

```
n_fft = 2 ** int(np.ceil(np.log2(2 * (len(h) - 1) / 0.01)))
```

This gives reasonable results for both the Hilbert and homomorphic methods, and gives the value used when `n_fft=None`.

Alternative implementations exist for creating minimum-phase filters, including zero inversion *[R251]* and spectral factorization *[R252]* *[R253]*. For more information, see:

http://dspguru.com/dsp/howtos/how-to-design-minimum-phase-fir-filters

References

[R250], *[R251]*, *[R252]*, *[R253]*, *[R254]*

Examples

Create an optimal linear-phase filter, then convert it to minimum phase:

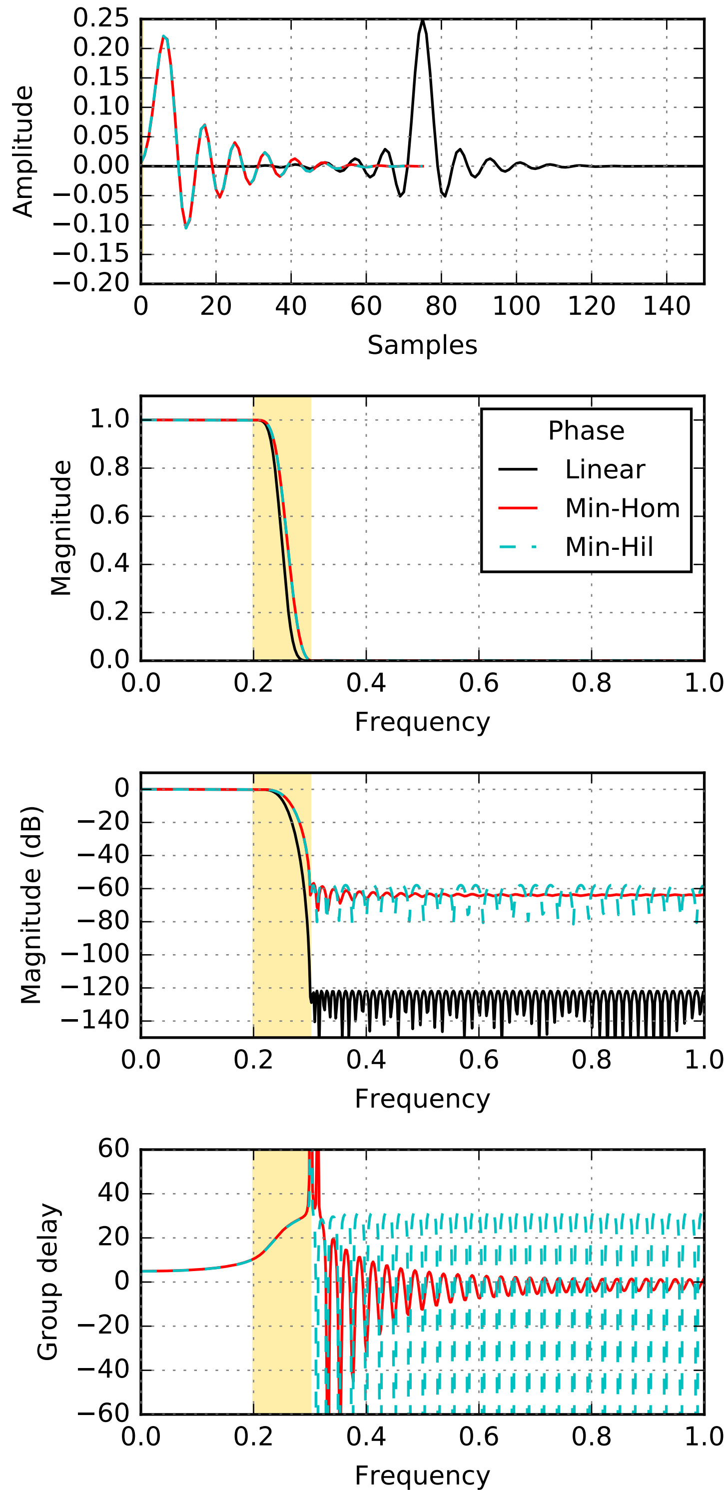
```
>>> from scipy.signal import remez, minimum_phase, freqz, group_delay
>>> import matplotlib.pyplot as plt
>>> freq = [0, 0.2, 0.3, 1.0]
>>> desired = [1, 0]
>>> h_linear = remez(151, freq, desired, Hz=2.)
```

Convert it to minimum phase:

```
>>> h_min_hom = minimum_phase(h_linear, method='homomorphic')
>>> h_min_hil = minimum_phase(h_linear, method='hilbert')
```

Compare the three filters:

```
>>> fig, axs = plt.subplots(4, figsize=(4, 8))
>>> for h, style, color in zip((h_linear, h_min_hom, h_min_hil),
...                             ('-', '-', '--'), ('k', 'r', 'c')):
...     w, H = freqz(h)
...     w, gd = group_delay((h, 1))
...     w /= np.pi
...     axs[0].plot(h, color=color, linestyle=style)
...     axs[1].plot(w, np.abs(H), color=color, linestyle=style)
...     axs[2].plot(w, 20 * np.log10(np.abs(H)), color=color, linestyle=style)
...     axs[3].plot(w, gd, color=color, linestyle=style)
>>> for ax in axs:
...     ax.grid(True, color='0.5')
...     ax.fill_between(freq[1:3], *ax.get_ylim(), color='#ffeeaa', zorder=1)
>>> axs[0].set(xlim=[0, len(h_linear) - 1], ylabel='Amplitude', xlabel='Samples')
>>> axs[1].legend(['Linear', 'Min-Hom', 'Min-Hil'], title='Phase')
>>> for ax, ylim in zip(axs[1:], ([0, 1.1], [-150, 10], [-60, 60])):
...     ax.set(xlim=[0, 1], ylim=ylim, xlabel='Frequency')
>>> axs[1].set(ylabel='Magnitude')
>>> axs[2].set(ylabel='Magnitude (dB)')
>>> axs[3].set(ylabel='Group delay')
>>> plt.tight_layout()
```



`scipy.signal.savgol_coeffs`(*window_length*, *polyorder*, *deriv=0*, *delta=1.0*, *pos=None*, *use='conv'*)

Compute the coefficients for a 1-d Savitzky-Golay FIR filter.

Parameters **window_length** : int

The length of the filter window (i.e. the number of coefficients). *win-*

dow_length must be an odd positive integer.

polyorder : int
The order of the polynomial used to fit the samples. *polyorder* must be less than *window_length*.

deriv : int, optional
The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.

delta : float, optional
The spacing of the samples to which the filter will be applied. This is only used if deriv > 0.

pos : int or None, optional
If pos is not None, it specifies evaluation position within the window. The default is the middle of the window.

use : str, optional
Either 'conv' or 'dot'. This argument chooses the order of the coefficients. The default is 'conv', which means that the coefficients are ordered to be used in a convolution. With use='dot', the order is reversed, so the filter is applied by dotting the coefficients with the data set.

Returns

coeffs : 1-d ndarray
The filter coefficients.

See also:

savgol_filter

Notes

New in version 0.14.0.

References

A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

Examples

```
>>> from scipy.signal import savgol_coeffs
>>> savgol_coeffs(5, 2)
array([-0.08571429,  0.34285714,  0.48571429,  0.34285714, -0.08571429])
>>> savgol_coeffs(5, 2, deriv=1)
array([  2.00000000e-01,   1.00000000e-01,   2.00607895e-16,
        -1.00000000e-01,  -2.00000000e-01])
```

Note that use='dot' simply reverses the coefficients.

```
>>> savgol_coeffs(5, 2, pos=3)
array([ 0.25714286,  0.37142857,  0.34285714,  0.17142857, -0.14285714])
>>> savgol_coeffs(5, 2, pos=3, use='dot')
array([-0.14285714,  0.17142857,  0.34285714,  0.37142857,  0.25714286])
```

x contains data from the parabola x = t**2, sampled at t = -1, 0, 1, 2, 3. *c* holds the coefficients that will compute the derivative at the last position. When dotted with *x* the result should be 6.

```
>>> x = np.array([1, 0, 1, 4, 9])
>>> c = savgol_coeffs(5, 2, pos=4, deriv=1, use='dot')
>>> c.dot(x)
6.0000000000000018
```

`scipy.signal.remez`(*numtaps*, *bands*, *desired*, *weight=None*, *Hz=1*, *type='bandpass'*, *maxiter=25*, *grid_density=16*)

Calculate the minimax optimal filter using the Remez exchange algorithm.

Calculate the filter-coefficients for the finite impulse response (FIR) filter whose transfer function minimizes the maximum error between the desired gain and the realized gain in the specified frequency bands using the Remez exchange algorithm.

Parameters

numtaps : int
: The desired number of taps in the filter. The number of taps is the number of terms in the filter, or the filter order plus one.

bands : array_like
: A monotonic sequence containing the band edges in Hz. All elements must be non-negative and less than half the sampling frequency as given by *Hz*.

desired : array_like
: A sequence half the size of bands containing the desired gain in each of the specified bands.

weight : array_like, optional
: A relative weighting to give to each band region. The length of *weight* has to be half the length of *bands*.

Hz : scalar, optional
: The sampling frequency in Hz. Default is 1.

type : {'bandpass', 'differentiator', 'hilbert'}, optional
: The type of filter:
  - 'bandpass' : flat response in bands. This is the default.
  - 'differentiator' : frequency proportional response in bands.
  - *'hilbert'* [filter with odd symmetry, that is, type III] (for even order) or type IV (for odd order) linear phase filters.

maxiter : int, optional
: Maximum number of iterations of the algorithm. Default is 25.

grid_density : int, optional
: Grid density. The dense grid used in `remez` is of size `(numtaps + 1) * grid_density`. Default is 16.

Returns

out : ndarray
: A rank-1 array containing the coefficients of the optimal (in a minimax sense) filter.

See also:

`firls`, `firwin`, `firwin2`, `minimum_phase`

References

[R260], *[R261]*

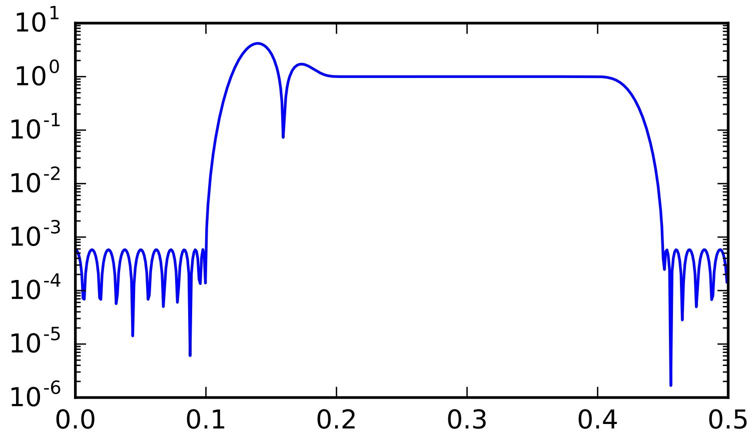
Examples

We want to construct a filter with a passband at 0.2-0.4 Hz, and stop bands at 0-0.1 Hz and 0.45-0.5 Hz. Note that this means that the behavior in the frequency ranges between those bands is unspecified and may overshoot.

```
>>> from scipy import signal
>>> bpass = signal.remez(72, [0, 0.1, 0.2, 0.4, 0.45, 0.5], [0, 1, 0])
>>> freq, response = signal.freqz(bpass)
>>> ampl = np.abs(response)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(111)
```

```
>>> ax1.semilogy(freq/(2*np.pi), ampl, 'b-')  # freq in Hz
>>> plt.show()
```



`scipy.signal.unique_roots(p, tol=0.001, rtype='min')`

Determine unique roots and their multiplicities from a list of roots.

Parameters **p** : array_like

The list of roots.

tol : float, optional

The tolerance for two roots to be considered equal. Default is 1e-3.

rtype : {'max', 'min, 'avg'}, optional

How to determine the returned root if multiple roots are within *tol* of each other.

- 'max': pick the maximum of those roots.
- 'min': pick the minimum of those roots.
- 'avg': take the average of those roots.

Returns **pout** : ndarray

The list of unique roots, sorted from low to high.

mult : ndarray

The multiplicity of each root.

Notes

This utility function is not specific to roots but can be used for any sequence of values for which uniqueness and multiplicity has to be determined. For a more general routine, see `numpy.unique`.

Examples

```
>>> from scipy import signal
>>> vals = [0, 1.3, 1.31, 2.8, 1.25, 2.2, 10.3]
>>> uniq, mult = signal.unique_roots(vals, tol=2e-2, rtype='avg')
```

Check which roots have multiplicity larger than 1:

```
>>> uniq[mult > 1]
array([ 1.305])
```

`scipy.signal.residue(b, a, tol=0.001, rtype='avg')`

Compute partial-fraction expansion of b(s) / a(s).

If *M* is the degree of numerator *b* and *N* the degree of denominator *a*:

```
        b(s)     b[0] s**(M) + b[1] s**(M-1) + ... + b[M]
H(s) = ------ = ------------------------------------------
        a(s)     a[0] s**(N) + a[1] s**(N-1) + ... + a[N]
```

then the partial-fraction expansion H(s) is defined as:

```
    r[0]       r[1]             r[-1]
= -------- + -------- + ... + --------- + k(s)
  (s-p[0])   (s-p[1])         (s-p[-1])
```

If there are any repeated roots (closer together than *tol*), then H(s) has terms like:

```
  r[i]      r[i+1]              r[i+n-1]
-------- + ----------- + ... + -----------
(s-p[i])  (s-p[i])**2          (s-p[i])**n
```

This function is used for polynomials in positive powers of s or z, such as analog filters or digital filters in controls engineering. For negative powers of z (typical for digital filters in DSP), use `residuez`.

Parameters
- **b** : array_like
  Numerator polynomial coefficients.
- **a** : array_like
  Denominator polynomial coefficients.

Returns
- **r** : ndarray
  Residues.
- **p** : ndarray
  Poles.
- **k** : ndarray
  Coefficients of the direct polynomial term.

See also:

`invres`, `residuez`, `numpy.poly`, `unique_roots`

`scipy.signal.`**`residuez`**(*b*, *a*, *tol=0.001*, *rtype='avg'*)

Compute partial-fraction expansion of b(z) / a(z).

If *M* is the degree of numerator *b* and *N* the degree of denominator *a*:

```
        b(z)     b[0] + b[1] z**(-1) + ... + b[M] z**(-M)
H(z) = ------ = ------------------------------------------
        a(z)     a[0] + a[1] z**(-1) + ... + a[N] z**(-N)
```

then the partial-fraction expansion H(z) is defined as:

```
        r[0]                   r[-1]
= --------------- + ... + ---------------- + k[0] + k[1]z**(-1) ...
  (1-p[0]z**(-1))         (1-p[-1]z**(-1))
```

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

```
     r[i]              r[i+1]                    r[i+n-1]
-------------- + ------------------ + ... + ------------------
(1-p[i]z**(-1))  (1-p[i]z**(-1))**2         (1-p[i]z**(-1))**n
```

This function is used for polynomials in negative powers of z, such as digital filters in DSP. For positive powers, use `residue`.

Parameters
- **b** : array_like
  Numerator polynomial coefficients.
- **a** : array_like
  Denominator polynomial coefficients.

Returns
- **r** : ndarray
  Residues.
- **p** : ndarray
  Poles.
- **k** : ndarray
  Coefficients of the direct polynomial term.

See also:

invresz, *residue*, *unique_roots*

`scipy.signal.invres(r, p, k, tol=0.001, rtype='avg')`

Compute b(s) and a(s) from partial fraction expansion.

If *M* is the degree of numerator *b* and *N* the degree of denominator *a*:

```
        b(s)     b[0] s**(M) + b[1] s**(M-1) + ... + b[M]
H(s) = ------ = ------------------------------------------
        a(s)     a[0] s**(N) + a[1] s**(N-1) + ... + a[N]
```

then the partial-fraction expansion H(s) is defined as:

```
    r[0]       r[1]             r[-1]
= -------- + -------- + ... + --------- + k(s)
  (s-p[0])   (s-p[1])         (s-p[-1])
```

If there are any repeated roots (closer together than *tol*), then H(s) has terms like:

```
  r[i]      r[i+1]              r[i+n-1]
-------- + ----------- + ... + -----------
(s-p[i])  (s-p[i])**2          (s-p[i])**n
```

This function is used for polynomials in positive powers of s or z, such as analog filters or digital filters in controls engineering. For negative powers of z (typical for digital filters in DSP), use *invresz*.

Parameters
- **r** : array_like
  Residues.
- **p** : array_like
  Poles.
- **k** : array_like
  Coefficients of the direct polynomial term.
- **tol** : float, optional
  The tolerance for two roots to be considered equal. Default is 1e-3.
- **rtype** : {'max', 'min, 'avg'}, optional
  How to determine the returned root if multiple roots are within *tol* of each other.
  - 'max': pick the maximum of those roots.
  - 'min': pick the minimum of those roots.
  - 'avg': take the average of those roots.

Returns
- **b** : ndarray
  Numerator polynomial coefficients.
- **a** : ndarray
  Denominator polynomial coefficients.

See also:

residue, *invresz*, *unique_roots*

scipy.signal.**invresz**(*r*, *p*, *k*, *tol=0.001*, *rtype='avg'*)

Compute b(z) and a(z) from partial fraction expansion.

If *M* is the degree of numerator *b* and *N* the degree of denominator *a*:

```
        b(z)     b[0] + b[1] z**(-1) + ... + b[M] z**(-M)
H(z) = ------ = ------------------------------------------
        a(z)     a[0] + a[1] z**(-1) + ... + a[N] z**(-N)
```

then the partial-fraction expansion H(z) is defined as:

```
        r[0]                   r[-1]
= --------------- + ... + ---------------- + k[0] + k[1]z**(-1) ...
  (1-p[0]z**(-1))         (1-p[-1]z**(-1))
```

If there are any repeated roots (closer than *tol*), then the partial fraction expansion has terms like:

```
     r[i]              r[i+1]                    r[i+n-1]
-------------- + ------------------ + ... + ------------------
(1-p[i]z**(-1))  (1-p[i]z**(-1))**2         (1-p[i]z**(-1))**n
```

This function is used for polynomials in negative powers of z, such as digital filters in DSP. For positive powers, use `invres`.

Parameters

r : array_like
: Residues.

p : array_like
: Poles.

k : array_like
: Coefficients of the direct polynomial term.

tol : float, optional
: The tolerance for two roots to be considered equal. Default is 1e-3.

rtype : {'max', 'min, 'avg'}, optional
: How to determine the returned root if multiple roots are within *tol* of each other.

- 'max': pick the maximum of those roots.
- 'min': pick the minimum of those roots.
- 'avg': take the average of those roots.

Returns

b : ndarray
: Numerator polynomial coefficients.

a : ndarray
: Denominator polynomial coefficients.

See also:

`residuez`, `unique_roots`, `invres`

exception scipy.signal.**BadCoefficients**

Warning about badly conditioned filter coefficients

Lower-level filter design functions:

`abcd_normalize`([A, B, C, D])	Check state-space matrices and ensure they are two-dimensional.
`band_stop_obj`(wp, ind, passb, stopb, gpass, ...)	Band Stop Objective Function for order minimization.
`besselap`(N[, norm])	Return (z,p,k) for analog prototype of an Nth-order Bessel filter.
`buttap`(N)	Return (z,p,k) for analog prototype of Nth-order Butterworth filter.

Continued on next page

Table 5.128 – continued from previous page

cheb1ap(N, rp)	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
cheb2ap(N, rs)	Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.
cmplx_sort(p)	Sort roots based on magnitude.
ellipap(N, rp, rs)	Return (z,p,k) of Nth-order elliptic analog lowpass filter.
lp2bp(b, a[, wo, bw])	Transform a lowpass filter prototype to a bandpass filter.
lp2bs(b, a[, wo, bw])	Transform a lowpass filter prototype to a bandstop filter.
lp2hp(b, a[, wo])	Transform a lowpass filter prototype to a highpass filter.
lp2lp(b, a[, wo])	Transform a lowpass filter prototype to a different frequency.
normalize(b, a)	Normalize numerator/denominator of a continuous-time transfer function.

scipy.signal.**abcd_normalize**(*A=None*, *B=None*, *C=None*, *D=None*)

Check state-space matrices and ensure they are two-dimensional.

If enough information on the system is provided, that is, enough properly-shaped arrays are passed to the function, the missing ones are built from this information, ensuring the correct number of rows and columns. Otherwise a ValueError is raised.

Parameters **A, B, C, D** : array_like, optional

State-space matrices. All of them are None (missing) by default. See *ss2tf* for format.

Returns **A, B, C, D** : array

Properly shaped state-space matrices.

Raises **ValueError**

If not enough information on the system was provided.

scipy.signal.**band_stop_obj**(*wp*, *ind*, *passb*, *stopb*, *gpass*, *gstop*, *type*)

Band Stop Objective Function for order minimization.

Returns the non-integer order for an analog band stop filter.

Parameters **wp** : scalar

Edge of passband *passb*.

ind : int, {0, 1}

Index specifying which *passb* edge to vary (0 or 1).

passb : ndarray

Two element sequence of fixed passband edges.

stopb : ndarray

Two element sequence of fixed stopband edges.

gstop : float

Amount of attenuation in stopband in dB.

gpass : float

Amount of ripple in the passband in dB.

type : {'butter', 'cheby', 'ellip'}

Type of filter.

Returns **n** : scalar

Filter order (possibly non-integer).

scipy.signal.**besselap**(*N*, *norm='phase'*)

Return (z,p,k) for analog prototype of an Nth-order Bessel filter.

Parameters **N** : int

The order of the filter.

norm : {'phase', 'delay', 'mag'}, optional

Frequency normalization:

phase The filter is normalized such that the phase response reaches its midpoint at an angular (e.g. rad/s) cutoff frequency of 1. This happens for both low-pass and high-pass filters, so this is the "phase-matched" case. *[R199]*

The magnitude response asymptotes are the same as a Butterworth filter of the same order with a cutoff of *Wn*.

This is the default, and matches MATLAB's implementation.

delay The filter is normalized such that the group delay in the passband is 1 (e.g. 1 second). This is the "natural" type obtained by solving Bessel polynomials

mag The filter is normalized such that the gain magnitude is -3 dB at angular frequency 1. This is called "frequency normalization" by Bond. *[R194]*

New in version 0.18.0.

Returns

z : ndarray

Zeros of the transfer function. Is always an empty array.

p : ndarray

Poles of the transfer function.

k : scalar

Gain of the transfer function. For phase-normalized, this is always 1.

See also:

bessel Filter design function using this prototype

Notes

To find the pole locations, approximate starting points are generated *[R195]* for the zeros of the ordinary Bessel polynomial *[R196]*, then the Aberth-Ehrlich method *[R197]* *[R198]* is used on the Kv(x) Bessel function to calculate more accurate zeros, and these locations are then inverted about the unit circle.

References

[R194], *[R195]*, *[R196]*, *[R197]*, *[R198]*, *[R199]*

scipy.signal.**buttap**(*N*)

Return (z,p,k) for analog prototype of Nth-order Butterworth filter.

The filter will have an angular (e.g. rad/s) cutoff frequency of 1.

See also:

butter Filter design function using this prototype

scipy.signal.**cheb1ap**(*N*, *rp*)

Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.

The returned filter prototype has *rp* decibels of ripple in the passband.

The filter's angular (e.g. rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first drops below `-rp`.

See also:

cheby1 Filter design function using this prototype

scipy.signal.**cheb2ap**(*N*, *rs*)

Return (z,p,k) for Nth-order Chebyshev type I analog lowpass filter.

The returned filter prototype has *rs* decibels of ripple in the stopband.

The filter's angular (e.g. rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first reaches `-rs`.

See also:

cheby2 Filter design function using this prototype

`scipy.signal.cmplx_sort(p)`

Sort roots based on magnitude.

Parameters **p** : array_like
The roots to sort, as a 1-D array.

Returns **p_sorted** : ndarray
Sorted roots.

indx : ndarray
Array of indices needed to sort the input *p*.

`scipy.signal.ellipap(N, rp, rs)`

Return (z,p,k) of Nth-order elliptic analog lowpass filter.

The filter is a normalized prototype that has *rp* decibels of ripple in the passband and a stopband *rs* decibels down.

The filter's angular (e.g. rad/s) cutoff frequency is normalized to 1, defined as the point at which the gain first drops below `-rp`.

See also:

ellip Filter design function using this prototype

References

[R215]

`scipy.signal.lp2bp(b, a, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandpass filter.

Return an analog band-pass filter with center frequency *wo* and bandwidth *bw* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

`scipy.signal.lp2bs(b, a, wo=1.0, bw=1.0)`

Transform a lowpass filter prototype to a bandstop filter.

Return an analog band-stop filter with center frequency *wo* and bandwidth *bw* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

`scipy.signal.lp2hp(b, a, wo=1.0)`

Transform a lowpass filter prototype to a highpass filter.

Return an analog high-pass filter with cutoff frequency *wo* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

`scipy.signal.lp2lp(b, a, wo=1.0)`

Transform a lowpass filter prototype to a different frequency.

Return an analog low-pass filter with cutoff frequency *wo* from an analog low-pass filter prototype with unity cutoff frequency, in transfer function ('ba') representation.

`scipy.signal.normalize(b, a)`

Normalize numerator/denominator of a continuous-time transfer function.

If values of *b* are too close to 0, they are removed. In that case, a BadCoefficients warning is emitted.

Parameters **b: array_like**

Numerator of the transfer function. Can be a 2d array to normalize multiple transfer functions.

a: array_like

Denominator of the transfer function. At most 1d.

Returns num: array

The numerator of the normalized transfer function. At least a 1d array. A 2d-array if the input *num* is a 2d array.

den: 1d-array

The denominator of the normalized transfer function.

Notes

Coefficients for both the numerator and denominator should be specified in descending exponent order (e.g., `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

5.20.5 Matlab-style IIR filter design

butter(N, Wn[, btype, analog, output])	Butterworth digital and analog filter design.
buttord(wp, ws, gpass, gstop[, analog])	Butterworth filter order selection.
cheby1(N, rp, Wn[, btype, analog, output])	Chebyshev type I digital and analog filter design.
cheb1ord(wp, ws, gpass, gstop[, analog])	Chebyshev type I filter order selection.
cheby2(N, rs, Wn[, btype, analog, output])	Chebyshev type II digital and analog filter design.
cheb2ord(wp, ws, gpass, gstop[, analog])	Chebyshev type II filter order selection.
ellip(N, rp, rs, Wn[, btype, analog, output])	Elliptic (Cauer) digital and analog filter design.
ellipord(wp, ws, gpass, gstop[, analog])	Elliptic (Cauer) filter order selection.
bessel(N, Wn[, btype, analog, output, norm])	Bessel/Thomson digital and analog filter design.
iirnotch(w0, Q)	Design second-order IIR notch digital filter.
iirpeak(w0, Q)	Design second-order IIR peak (resonant) digital filter.

scipy.signal.**butter**(*N*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*)

Butterworth digital and analog filter design.

Design an Nth-order digital or analog Butterworth filter and return the filter coefficients.

Parameters **N** : int

The order of the filter.

Wn : array_like

A scalar or length-2 sequence giving the critical frequencies. For a Butterworth filter, this is the point at which the gain drops to 1/sqrt(2) that of the passband (the "-3 dB point"). For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.) For analog filters, *Wn* is an angular frequency (e.g. rad/s).

btype : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional

The type of filter. Default is 'lowpass'.

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

output : {'ba', 'zpk', 'sos'}, optional

Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns **b, a** : ndarray, ndarray

Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float

Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray

Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

buttord, *buttap*

Notes

The Butterworth filter has maximally flat frequency response in the passband.

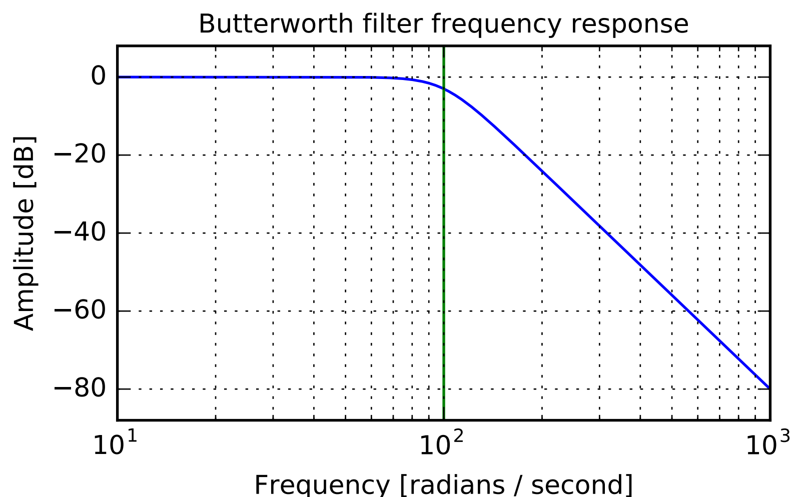
The `'sos'` output parameter was added in 0.16.0.

Examples

Plot the filter's frequency response, showing the critical points:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.butter(4, 100, 'low', analog=True)
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Butterworth filter frequency response')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.axvline(100, color='green') # cutoff frequency
>>> plt.show()
```



`scipy.signal.`**`buttord`**(*wp*, *ws*, *gpass*, *gstop*, *analog=False*)

Butterworth filter order selection.

Return the order of the lowest order digital or analog Butterworth filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

Parameters **wp, ws** : float

Passband and stopband edge frequencies. For digital filters, these are normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: wp = 0.2, ws = 0.3
- Highpass: wp = 0.3, ws = 0.2
- Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6]
- Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g. rad/s).

gpass : float

The maximum loss in the passband (dB).

gstop : float

The minimum attenuation in the stopband (dB).

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

Returns **ord** : int

The lowest order for a Butterworth filter which meets specs.

wn : ndarray or float

The Butterworth natural frequency (i.e. the "3dB frequency"). Should be used with `butter` to give filter results.

See also:

butter Filter design using order and critical points
cheb1ord Find order and critical points from passband and stopband spec

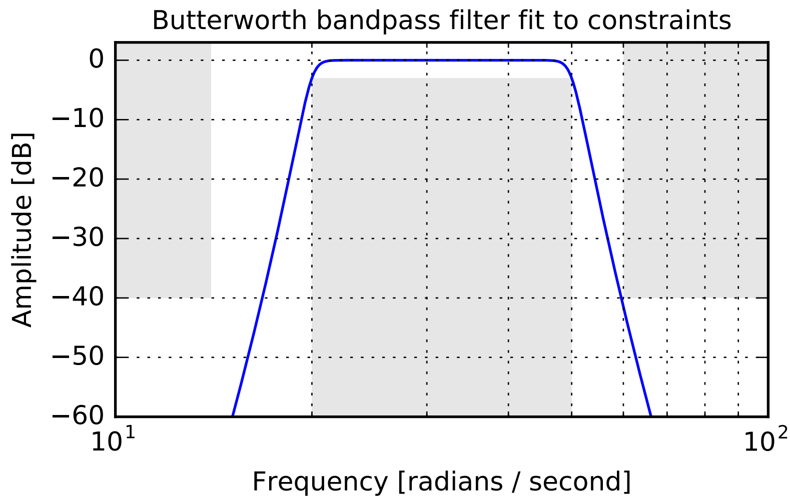
cheb2ord, *ellipord*
iirfilter General filter design using order and critical frequencies
iirdesign General filter design using passband and stopband spec

Examples

Design an analog bandpass filter with passband within 3 dB from 20 to 50 rad/s, while rejecting at least -40 dB below 14 and above 60 rad/s. Plot its frequency response, showing the passband and stopband constraints in gray.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> N, Wn = signal.buttord([20, 50], [14, 60], 3, 40, True)
>>> b, a = signal.butter(N, Wn, 'band', True)
>>> w, h = signal.freqs(b, a, np.logspace(1, 2, 500))
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Butterworth bandpass filter fit to constraints')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.grid(which='both', axis='both')
>>> plt.fill([1,  14,  14,   1], [-40, -40, 99, 99], '0.9', lw=0) # stop
>>> plt.fill([20, 20,  50,  50], [-99, -3, -3, -99], '0.9', lw=0) # pass
>>> plt.fill([60, 60, 1e9, 1e9], [99, -40, -40, 99], '0.9', lw=0) # stop
>>> plt.axis([10, 100, -60, 3])
>>> plt.show()
```



`scipy.signal.cheby1`(*N*, *rp*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*)

Chebyshev type I digital and analog filter design.

Design an Nth-order digital or analog Chebyshev type I filter and return the filter coefficients.

Parameters

N : int
The order of the filter.

rp : float
The maximum ripple allowed below unity gain in the passband. Specified in decibels, as a positive number.

Wn : array_like
A scalar or length-2 sequence giving the critical frequencies. For Type I filters, this is the point in the transition band at which the gain first drops below -*rp*. For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.) For analog filters, *Wn* is an angular frequency (e.g. rad/s).

btype : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional
The type of filter. Default is 'lowpass'.

analog : bool, optional
When True, return an analog filter, otherwise a digital filter is returned.

output : {'ba', 'zpk', 'sos'}, optional
Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns

b, a : ndarray, ndarray
Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float
Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray
Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

cheb1ord, *cheb1ap*

Notes

The Chebyshev type I filter maximizes the rate of cutoff between the frequency response's passband and stopband, at the expense of ripple in the passband and increased ringing in the step response.

Type I filters roll off faster than Type II (*cheby2*), but Type II filters do not have any ripple in the passband.

The equiripple passband has N maxima or minima (for example, a 5th-order filter has 3 maxima and 2 minima). Consequently, the DC gain is unity for odd-order filters, or -rp dB for even-order filters.

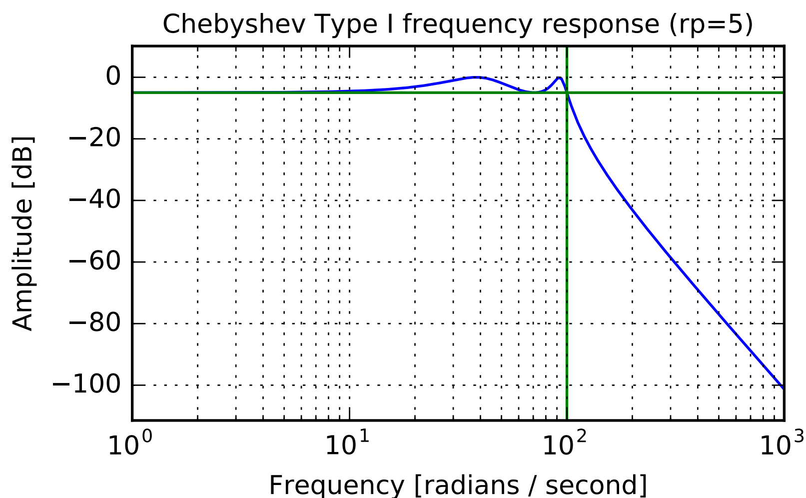
The `'sos'` output parameter was added in 0.16.0.

Examples

Plot the filter's frequency response, showing the critical points:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.cheby1(4, 5, 100, 'low', analog=True)
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Chebyshev Type I frequency response (rp=5)')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.axvline(100, color='green') # cutoff frequency
>>> plt.axhline(-5, color='green') # rp
>>> plt.show()
```



`scipy.signal.`**`cheb1ord`**(*wp*, *ws*, *gpass*, *gstop*, *analog=False*)

Chebyshev type I filter order selection.

Return the order of the lowest order digital or analog Chebyshev Type I filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

Parameters **wp, ws** : float

Passband and stopband edge frequencies. For digital filters, these are normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: wp = 0.2, ws = 0.3
- Highpass: wp = 0.3, ws = 0.2
- Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6]
- Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g. rad/s).

gpass : float
The maximum loss in the passband (dB).

gstop : float
The minimum attenuation in the stopband (dB).

analog : bool, optional
When True, return an analog filter, otherwise a digital filter is returned.

Returns

ord : int
The lowest order for a Chebyshev type I filter that meets specs.

wn : ndarray or float
The Chebyshev natural frequency (the "3dB frequency") for use with *cheby1* to give filter results.

See also:

cheby1 Filter design using order and critical points
buttord Find order and critical points from passband and stopband spec

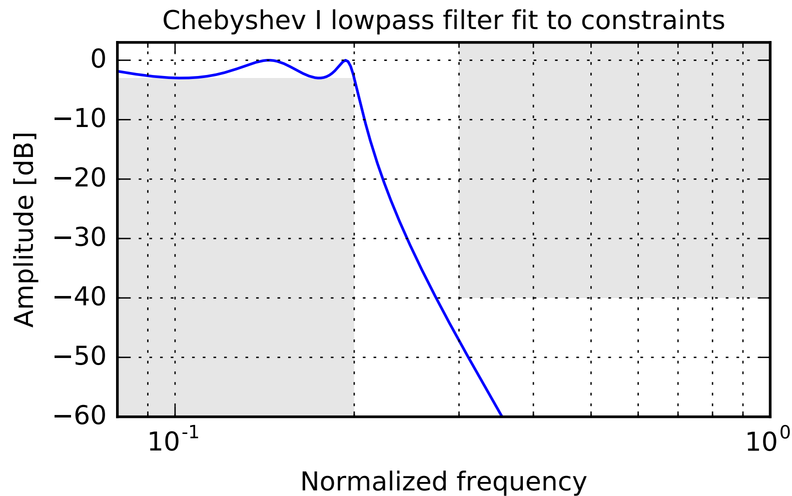
cheb2ord, *ellipord*
iirfilter General filter design using order and critical frequencies
iirdesign General filter design using passband and stopband spec

Examples

Design a digital lowpass filter such that the passband is within 3 dB up to 0.2*(fs/2), while rejecting at least -40 dB above 0.3*(fs/2). Plot its frequency response, showing the passband and stopband constraints in gray.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> N, Wn = signal.cheb1ord(0.2, 0.3, 3, 40)
>>> b, a = signal.cheby1(N, 3, Wn, 'low')
>>> w, h = signal.freqz(b, a)
>>> plt.semilogx(w / np.pi, 20 * np.log10(abs(h)))
>>> plt.title('Chebyshev I lowpass filter fit to constraints')
>>> plt.xlabel('Normalized frequency')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.grid(which='both', axis='both')
>>> plt.fill([.01, 0.2, 0.2, .01], [-3, -3, -99, -99], '0.9', lw=0) # stop
>>> plt.fill([0.3, 0.3,   2,   2], [ 9, -40, -40,   9], '0.9', lw=0) # pass
>>> plt.axis([0.08, 1, -60, 3])
>>> plt.show()
```



`scipy.signal.cheby2`(*N*, *rs*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*)

Chebyshev type II digital and analog filter design.

Design an Nth-order digital or analog Chebyshev type II filter and return the filter coefficients.

Parameters

N : int
: The order of the filter.

rs : float
: The minimum attenuation required in the stop band. Specified in decibels, as a positive number.

Wn : array_like
: A scalar or length-2 sequence giving the critical frequencies. For Type II filters, this is the point in the transition band at which the gain first reaches -*rs*. For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.) For analog filters, *Wn* is an angular frequency (e.g. rad/s).

btype : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional
: The type of filter. Default is 'lowpass'.

analog : bool, optional
: When True, return an analog filter, otherwise a digital filter is returned.

output : {'ba', 'zpk', 'sos'}, optional
: Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns

b, a : ndarray, ndarray
: Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float
: Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray
: Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

cheb2ord, *cheb2ap*

Notes

The Chebyshev type II filter maximizes the rate of cutoff between the frequency response's passband and stopband, at the expense of ripple in the stopband and increased ringing in the step response.

Type II filters do not roll off as fast as Type I (`cheby1`).

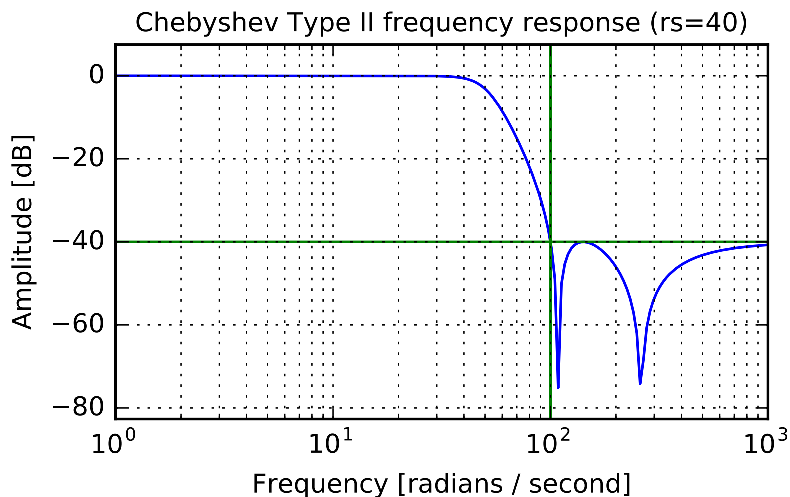
The `'sos'` output parameter was added in 0.16.0.

Examples

Plot the filter's frequency response, showing the critical points:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.cheby2(4, 40, 100, 'low', analog=True)
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Chebyshev Type II frequency response (rs=40)')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.axvline(100, color='green') # cutoff frequency
>>> plt.axhline(-40, color='green') # rs
>>> plt.show()
```



`scipy.signal.`**`cheb2ord`**(*wp*, *ws*, *gpass*, *gstop*, *analog=False*)

Chebyshev type II filter order selection.

Return the order of the lowest order digital or analog Chebyshev Type II filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

Parameters **wp, ws** : float

Passband and stopband edge frequencies. For digital filters, these are normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: wp = 0.2, ws = 0.3
- Highpass: wp = 0.3, ws = 0.2

- Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6]
- Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g. rad/s).

gpass : float

The maximum loss in the passband (dB).

gstop : float

The minimum attenuation in the stopband (dB).

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

Returns

ord : int

The lowest order for a Chebyshev type II filter that meets specs.

wn : ndarray or float

The Chebyshev natural frequency (the "3dB frequency") for use with `cheby2` to give filter results.

See also:

`cheby2` Filter design using order and critical points

`buttord` Find order and critical points from passband and stopband spec

`cheb1ord`, `ellipord`

`iirfilter` General filter design using order and critical frequencies

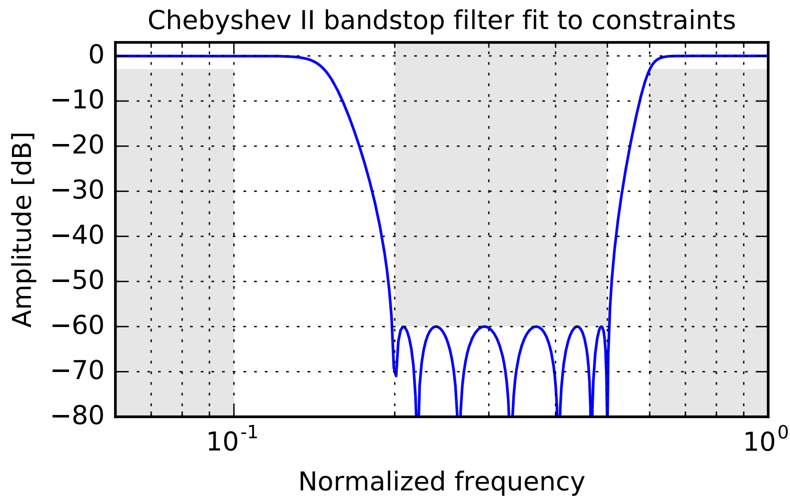
`iirdesign` General filter design using passband and stopband spec

Examples

Design a digital bandstop filter which rejects -60 dB from 0.2*(fs/2) to 0.5*(fs/2), while staying within 3 dB below 0.1*(fs/2) or above 0.6*(fs/2). Plot its frequency response, showing the passband and stopband constraints in gray.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> N, Wn = signal.cheb2ord([0.1, 0.6], [0.2, 0.5], 3, 60)
>>> b, a = signal.cheby2(N, 60, Wn, 'stop')
>>> w, h = signal.freqz(b, a)
>>> plt.semilogx(w / np.pi, 20 * np.log10(abs(h)))
>>> plt.title('Chebyshev II bandstop filter fit to constraints')
>>> plt.xlabel('Normalized frequency')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.grid(which='both', axis='both')
>>> plt.fill([.01, .1, .1, .01], [-3,  -3, -99, -99], '0.9', lw=0) # stop
>>> plt.fill([.2,  .2, .5,  .5], [ 9, -60, -60,   9], '0.9', lw=0) # pass
>>> plt.fill([.6,  .6,  2,   2], [-99, -3,  -3, -99], '0.9', lw=0) # stop
>>> plt.axis([0.06, 1, -80, 3])
>>> plt.show()
```



`scipy.signal.ellip`(*N*, *rp*, *rs*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*)

Elliptic (Cauer) digital and analog filter design.

Design an Nth-order digital or analog elliptic filter and return the filter coefficients.

Parameters

N : int
: The order of the filter.

rp : float
: The maximum ripple allowed below unity gain in the passband. Specified in decibels, as a positive number.

rs : float
: The minimum attenuation required in the stop band. Specified in decibels, as a positive number.

Wn : array_like
: A scalar or length-2 sequence giving the critical frequencies. For elliptic filters, this is the point in the transition band at which the gain first drops below *-rp*. For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.) For analog filters, *Wn* is an angular frequency (e.g. rad/s).

btype : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional
: The type of filter. Default is 'lowpass'.

analog : bool, optional
: When True, return an analog filter, otherwise a digital filter is returned.

output : {'ba', 'zpk', 'sos'}, optional
: Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

Returns

b, a : ndarray, ndarray
: Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float
: Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray
: Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

See also:

`ellipord`, `ellipap`

Notes

Also known as Cauer or Zolotarev filters, the elliptical filter maximizes the rate of transition between the frequency response's passband and stopband, at the expense of ripple in both, and increased ringing in the step response.

As *rp* approaches 0, the elliptical filter becomes a Chebyshev type II filter (`cheby2`). As *rs* approaches 0, it becomes a Chebyshev type I filter (`cheby1`). As both approach 0, it becomes a Butterworth filter (`butter`).

The equiripple passband has N maxima or minima (for example, a 5th-order filter has 3 maxima and 2 minima). Consequently, the DC gain is unity for odd-order filters, or -rp dB for even-order filters.

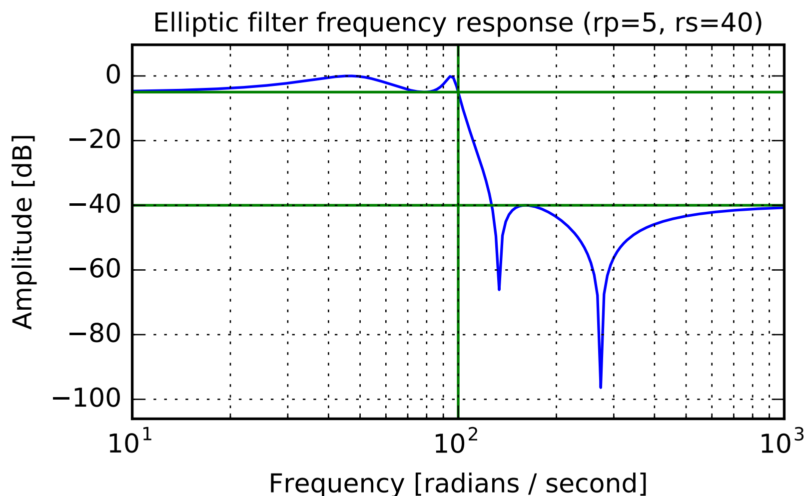
The `'sos'` output parameter was added in 0.16.0.

Examples

Plot the filter's frequency response, showing the critical points:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.ellip(4, 5, 40, 100, 'low', analog=True)
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Elliptic filter frequency response (rp=5, rs=40)')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.axvline(100, color='green') # cutoff frequency
>>> plt.axhline(-40, color='green') # rs
>>> plt.axhline(-5, color='green') # rp
>>> plt.show()
```



`scipy.signal.`**`ellipord`**(*wp*, *ws*, *gpass*, *gstop*, *analog=False*)
Elliptic (Cauer) filter order selection.

Return the order of the lowest order digital or analog elliptic filter that loses no more than *gpass* dB in the passband and has at least *gstop* dB attenuation in the stopband.

Parameters

wp, ws : float

Passband and stopband edge frequencies. For digital filters, these are normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*wp* and *ws* are thus in half-cycles / sample.) For example:

- Lowpass: wp = 0.2, ws = 0.3
- Highpass: wp = 0.3, ws = 0.2
- Bandpass: wp = [0.2, 0.5], ws = [0.1, 0.6]
- Bandstop: wp = [0.1, 0.6], ws = [0.2, 0.5]

For analog filters, *wp* and *ws* are angular frequencies (e.g. rad/s).

gpass : float

The maximum loss in the passband (dB).

gstop : float

The minimum attenuation in the stopband (dB).

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned.

Returns

ord : int

The lowest order for an Elliptic (Cauer) filter that meets specs.

wn : ndarray or float

The Chebyshev natural frequency (the "3dB frequency") for use with `ellip` to give filter results.

See also:

ellip Filter design using order and critical points

buttord Find order and critical points from passband and stopband spec

cheb1ord, *cheb2ord*

iirfilter General filter design using order and critical frequencies

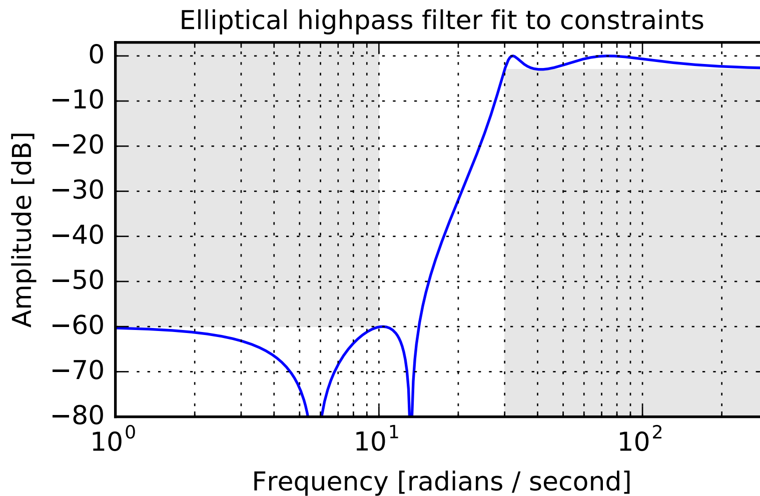
iirdesign General filter design using passband and stopband spec

Examples

Design an analog highpass filter such that the passband is within 3 dB above 30 rad/s, while rejecting -60 dB at 10 rad/s. Plot its frequency response, showing the passband and stopband constraints in gray.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> N, Wn = signal.ellipord(30, 10, 3, 60, True)
>>> b, a = signal.ellip(N, 3, 60, Wn, 'high', True)
>>> w, h = signal.freqs(b, a, np.logspace(0, 3, 500))
>>> plt.semilogx(w, 20 * np.log10(abs(h)))
>>> plt.title('Elliptical highpass filter fit to constraints')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.grid(which='both', axis='both')
>>> plt.fill([.1, 10,  10,  .1], [1e4, 1e4, -60, -60], '0.9', lw=0) # stop
>>> plt.fill([30, 30, 1e9, 1e9], [-99,  -3,  -3, -99], '0.9', lw=0) # pass
>>> plt.axis([1, 300, -80, 3])
>>> plt.show()
```



`scipy.signal.bessel`(*N*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*, *norm='phase'*)

Bessel/Thomson digital and analog filter design.

Design an Nth-order digital or analog Bessel filter and return the filter coefficients.

Parameters

N : int

The order of the filter.

Wn : array_like

A scalar or length-2 sequence giving the critical frequencies (defined by the *norm* parameter). For analog filters, *Wn* is an angular frequency (e.g. rad/s). For digital filters, *Wn* is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (*Wn* is thus in half-cycles / sample.)

btype : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional

The type of filter. Default is 'lowpass'.

analog : bool, optional

When True, return an analog filter, otherwise a digital filter is returned. (See Notes.)

output : {'ba', 'zpk', 'sos'}, optional

Type of output: numerator/denominator ('ba'), pole-zero ('zpk'), or second-order sections ('sos'). Default is 'ba'.

norm : {'phase', 'delay', 'mag'}, optional

Critical frequency normalization:

phase The filter is normalized such that the phase response reaches its midpoint at angular (e.g. rad/s) frequency *Wn*. This happens for both low-pass and high-pass filters, so this is the "phase-matched" case.

The magnitude response asymptotes are the same as a Butterworth filter of the same order with a cutoff of *Wn*.

This is the default, and matches MATLAB's implementation.

delay The filter is normalized such that the group delay in the passband is 1/*Wn* (e.g. seconds). This is the "natural" type obtained by solving Bessel polynomials.

mag The filter is normalized such that the gain magnitude is -3 dB at angular frequency *Wn*.

New in version 0.18.0.

Returns

b, a : ndarray, ndarray

Numerator (b) and denominator (a) polynomials of the IIR filter. Only returned if `output='ba'`.

z, p, k : ndarray, ndarray, float

Zeros, poles, and system gain of the IIR filter transfer function. Only returned if `output='zpk'`.

sos : ndarray

Second-order sections representation of the IIR filter. Only returned if `output=='sos'`.

Notes

Also known as a Thomson filter, the analog Bessel filter has maximally flat group delay and maximally linear phase response, with very little ringing in the step response. *[R193]*

The Bessel is inherently an analog filter. This function generates digital Bessel filters using the bilinear transform, which does not preserve the phase response of the analog filter. As such, it is only approximately correct at frequencies below about fs/4. To get maximally-flat group delay at higher frequencies, the analog Bessel filter must be transformed using phase-preserving techniques.

See *besselap* for implementation details and references.

The `'sos'` output parameter was added in 0.16.0.

References

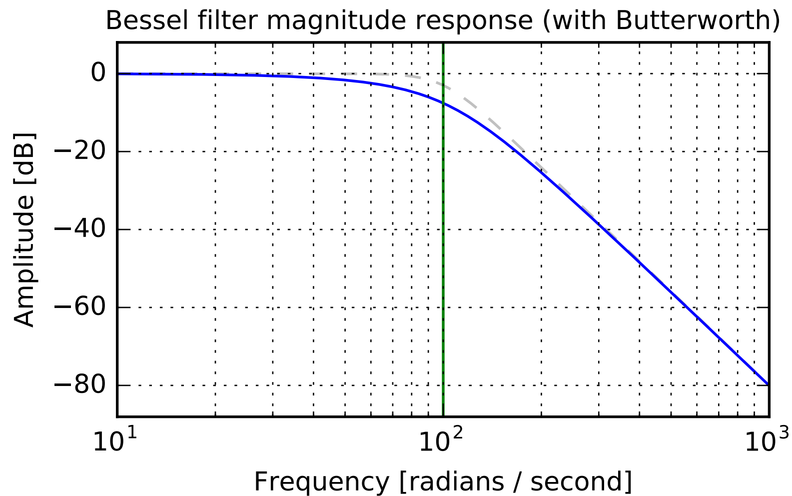
[R193]

Examples

Plot the phase-normalized frequency response, showing the relationship to the Butterworth's cutoff frequency (green):

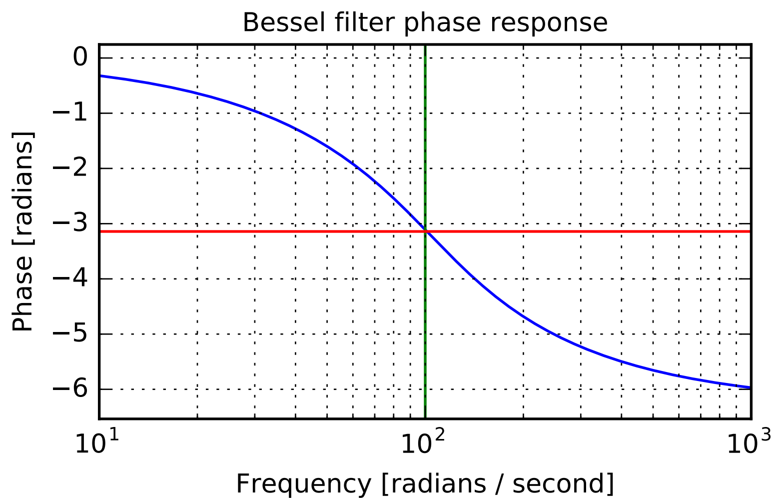
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> b, a = signal.butter(4, 100, 'low', analog=True)
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(np.abs(h)), color='silver', ls='dashed')
>>> b, a = signal.bessel(4, 100, 'low', analog=True, norm='phase')
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(np.abs(h)))
>>> plt.title('Bessel filter magnitude response (with Butterworth)')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.axvline(100, color='green')  # cutoff frequency
>>> plt.show()
```



and the phase midpoint:

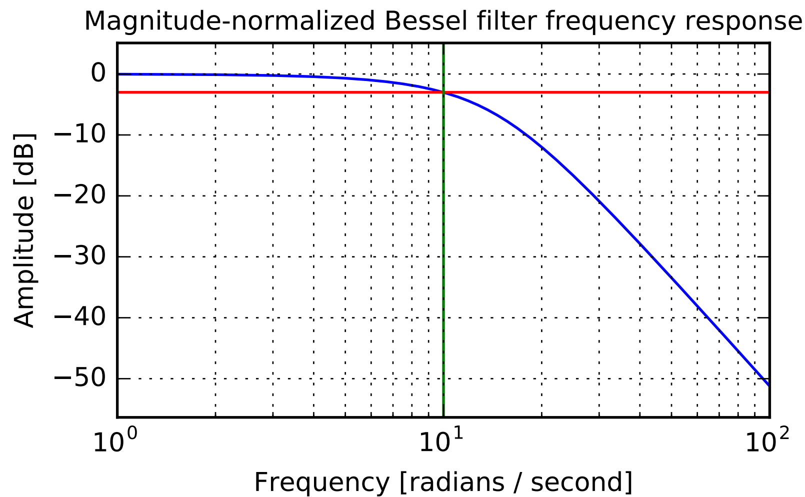
```
>>> plt.figure()
>>> plt.semilogx(w, np.unwrap(np.angle(h)))
>>> plt.axvline(100, color='green')  # cutoff frequency
>>> plt.axhline(-np.pi, color='red')  # phase midpoint
>>> plt.title('Bessel filter phase response')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Phase [radians]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.show()
```



Plot the magnitude-normalized frequency response, showing the -3 dB cutoff:

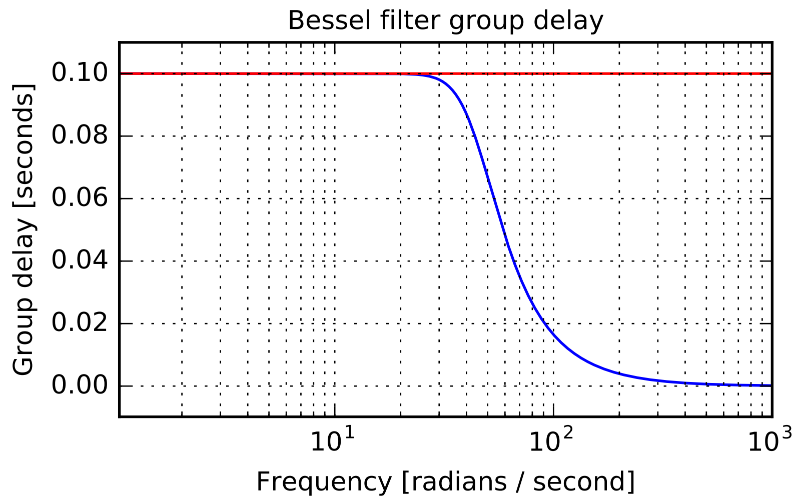
```
>>> b, a = signal.bessel(3, 10, 'low', analog=True, norm='mag')
>>> w, h = signal.freqs(b, a)
>>> plt.semilogx(w, 20 * np.log10(np.abs(h)))
```

```
>>> plt.axhline(-3, color='red')  # -3 dB magnitude
>>> plt.axvline(10, color='green')  # cutoff frequency
>>> plt.title('Magnitude-normalized Bessel filter frequency response')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Amplitude [dB]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.show()
```



Plot the delay-normalized filter, showing the maximally-flat group delay at 0.1 seconds:

```
>>> b, a = signal.bessel(5, 1/0.1, 'low', analog=True, norm='delay')
>>> w, h = signal.freqs(b, a)
>>> plt.figure()
>>> plt.semilogx(w[1:], -np.diff(np.unwrap(np.angle(h)))/np.diff(w))
>>> plt.axhline(0.1, color='red')  # 0.1 seconds group delay
>>> plt.title('Bessel filter group delay')
>>> plt.xlabel('Frequency [radians / second]')
>>> plt.ylabel('Group delay [seconds]')
>>> plt.margins(0, 0.1)
>>> plt.grid(which='both', axis='both')
>>> plt.show()
```



`scipy.signal.iirnotch(w0, Q)`

Design second-order IIR notch digital filter.

A notch filter is a band-stop filter with a narrow bandwidth (high quality factor). It rejects a narrow frequency band and leaves the rest of the spectrum little changed.

Parameters **w0** : float

Normalized frequency to remove from a signal. It is a scalar that must satisfy `0 < w0 < 1`, with `w0 = 1` corresponding to half of the sampling frequency.

Q : float

Quality factor. Dimensionless parameter that characterizes notch filter -3 dB bandwidth `bw` relative to its center frequency, `Q = w0/bw`.

Returns **b, a** : ndarray, ndarray

Numerator (`b`) and denominator (`a`) polynomials of the IIR filter.

See also:

iirpeak

Notes

References

[R239]

Examples

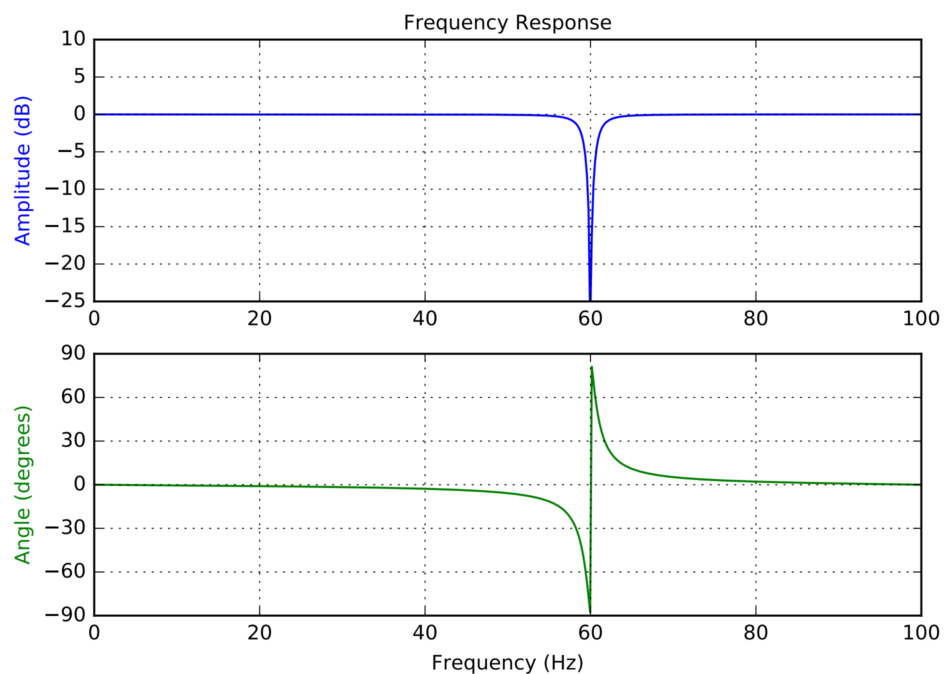
Design and plot filter to remove the 60Hz component from a signal sampled at 200Hz, using a quality factor Q = 30

```
>>> from scipy import signal
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

```
>>> fs = 200.0  # Sample frequency (Hz)
>>> f0 = 60.0  # Frequency to be removed from signal (Hz)
>>> Q = 30.0  # Quality factor
>>> w0 = f0/(fs/2)  # Normalized Frequency
```

```
>>> # Design notch filter
>>> b, a = signal.iirnotch(w0, Q)
```

```
>>> # Frequency response
>>> w, h = signal.freqz(b, a)
>>> # Generate frequency axis
>>> freq = w*fs/(2*np.pi)
>>> # Plot
>>> fig, ax = plt.subplots(2, 1, figsize=(8, 6))
>>> ax[0].plot(freq, 20*np.log10(abs(h)), color='blue')
>>> ax[0].set_title("Frequency Response")
>>> ax[0].set_ylabel("Amplitude (dB)", color='blue')
>>> ax[0].set_xlim([0, 100])
>>> ax[0].set_ylim([-25, 10])
>>> ax[0].grid()
>>> ax[1].plot(freq, np.unwrap(np.angle(h))*180/np.pi, color='green')
>>> ax[1].set_ylabel("Angle (degrees)", color='green')
>>> ax[1].set_xlabel("Frequency (Hz)")
>>> ax[1].set_xlim([0, 100])
>>> ax[1].set_yticks([-90, -60, -30, 0, 30, 60, 90])
>>> ax[1].set_ylim([-90, 90])
>>> ax[1].grid()
>>> plt.show()
```



`scipy.signal.`**`iirpeak`**(*w0*, *Q*)

Design second-order IIR peak (resonant) digital filter.

A peak filter is a band-pass filter with a narrow bandwidth (high quality factor). It rejects components outside a narrow frequency band.

Parameters

w0 : float

Normalized frequency to be retained in a signal. It is a scalar that must satisfy `0 < w0 < 1`, with `w0 = 1` corresponding to half of the sampling frequency.

Q : float

Quality factor. Dimensionless parameter that characterizes peak filter -3 dB bandwidth `bw` relative to its center frequency, `Q = w0/bw`.

Returns

b, a : ndarray, ndarray

Numerator (`b`) and denominator (`a`) polynomials of the IIR filter.

See also:

`iirnotch`

Notes

References

[R240]

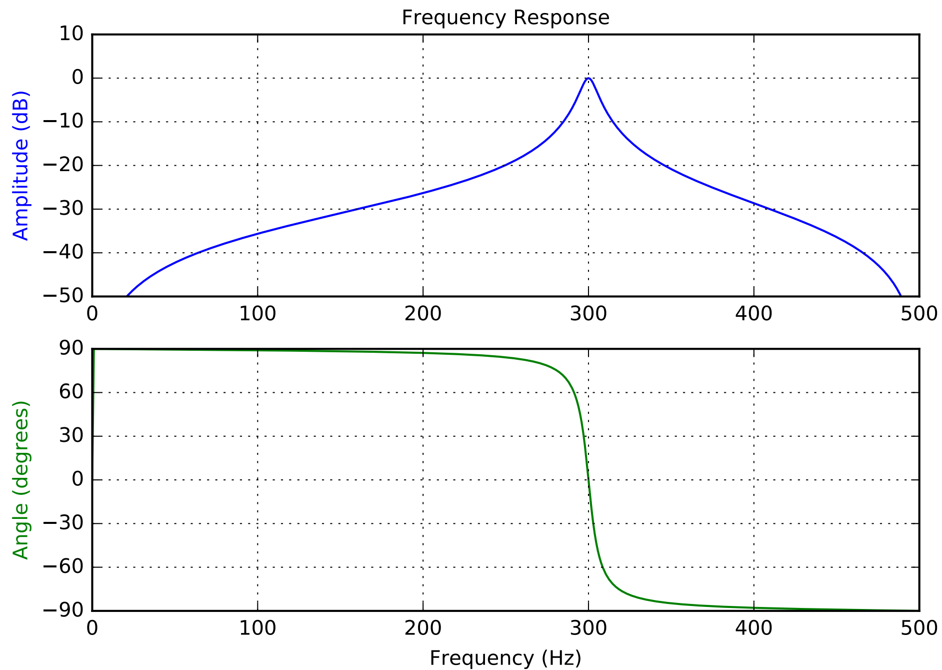
Examples

Design and plot filter to remove the frequencies other than the 300Hz component from a signal sampled at 1000Hz, using a quality factor Q = 30

```
>>> from scipy import signal
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

```
>>> fs = 1000.0  # Sample frequency (Hz)
>>> f0 = 300.0  # Frequency to be retained (Hz)
>>> Q = 30.0  # Quality factor
>>> w0 = f0/(fs/2)  # Normalized Frequency
>>> # Design peak filter
>>> b, a = signal.iirpeak(w0, Q)
```

```
>>> # Frequency response
>>> w, h = signal.freqz(b, a)
>>> # Generate frequency axis
>>> freq = w*fs/(2*np.pi)
>>> # Plot
>>> fig, ax = plt.subplots(2, 1, figsize=(8, 6))
>>> ax[0].plot(freq, 20*np.log10(abs(h)), color='blue')
>>> ax[0].set_title("Frequency Response")
>>> ax[0].set_ylabel("Amplitude (dB)", color='blue')
>>> ax[0].set_xlim([0, 500])
>>> ax[0].set_ylim([-50, 10])
>>> ax[0].grid()
>>> ax[1].plot(freq, np.unwrap(np.angle(h))*180/np.pi, color='green')
>>> ax[1].set_ylabel("Angle (degrees)", color='green')
>>> ax[1].set_xlabel("Frequency (Hz)")
>>> ax[1].set_xlim([0, 500])
>>> ax[1].set_yticks([-90, -60, -30, 0, 30, 60, 90])
>>> ax[1].set_ylim([-90, 90])
>>> ax[1].grid()
>>> plt.show()
```



5.20.6 Continuous-Time Linear Systems

lti(*system)	Continuous-time linear time invariant system base class.
StateSpace(*system, **kwargs)	Linear Time Invariant system in state-space form.
TransferFunction(*system, **kwargs)	Linear Time Invariant system class in transfer function form.
ZerosPolesGain(*system, **kwargs)	Linear Time Invariant system class in zeros, poles, gain form.
lsim(system, U, T[, X0, interp])	Simulate output of a continuous-time linear system.
lsim2(system[, U, T, X0])	Simulate output of a continuous-time linear system, by using the ODE solver *scipy.integrate.odeint*.
impulse(system[, X0, T, N])	Impulse response of continuous-time system.
impulse2(system[, X0, T, N])	Impulse response of a single-input, continuous-time linear system.
step(system[, X0, T, N])	Step response of continuous-time system.
step2(system[, X0, T, N])	Step response of continuous-time system.
freqresp(system[, w, n])	Calculate the frequency response of a continuous-time system.
bode(system[, w, n])	Calculate Bode magnitude and phase data of a continuous-time system.

class `scipy.signal.lti`(**system*)

Continuous-time linear time invariant system base class.

Parameters ***system** : arguments

The *lti* class can be instantiated with either 2, 3 or 4 arguments. The following gives the number of arguments and the corresponding continuous-time subclass that is created:

- 2: *TransferFunction*: (numerator, denominator)
- 3: *ZerosPolesGain*: (zeros, poles, gain)
- 4: *StateSpace*: (A, B, C, D)

Each argument can be an array or a sequence.

See also:

ZerosPolesGain, *StateSpace*, *TransferFunction*, *dlti*

Notes

lti instances do not exist directly. Instead, *lti* creates an instance of one of its subclasses: *StateSpace*, *TransferFunction* or *ZerosPolesGain*.

If (numerator, denominator) is passed in for `*system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g., `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

Changing the value of properties that are not directly part of the current system representation (such as the *zeros* of a *StateSpace* system) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the zeros, poles or gain.

Examples

```
>>> from scipy import signal
```

```
>>> signal.lti(1, 2, 3, 4)
StateSpaceContinuous(
array([[1]]),
array([[2]]),
array([[3]]),
array([[4]]),
dt: None
)
```

```
>>> signal.lti([1, 2], [3, 4], 5)
ZerosPolesGainContinuous(
array([1, 2]),
array([3, 4]),
5,
dt: None
)
```

```
>>> signal.lti([3, 4], [1, 2])
TransferFunctionContinuous(
array([ 3.,  4.]),
array([ 1.,  2.]),
dt: None
)
```

Attributes

A	State matrix of the *StateSpace* system.
B	Input matrix of the *StateSpace* system.
C	Output matrix of the *StateSpace* system.
D	Feedthrough matrix of the *StateSpace* system.
den	Denominator of the *TransferFunction* system.
dt	Return the sampling time of the system, *None* for `lti` systems.
gain	Gain of the *ZerosPolesGain* system.
num	Numerator of the *TransferFunction* system.
poles	Poles of the system.
zeros	Zeros of the system.

lti.**A**
State matrix of the *StateSpace* system.

lti.**B**
Input matrix of the *StateSpace* system.

lti.**C**
Output matrix of the *StateSpace* system.

lti.**D**
Feedthrough matrix of the *StateSpace* system.

lti.**den**
Denominator of the *TransferFunction* system.

lti.**dt**
Return the sampling time of the system, *None* for `lti` systems.

lti.**gain**
Gain of the *ZerosPolesGain* system.

lti.**num**
Numerator of the *TransferFunction* system.

lti.**poles**
Poles of the system.

lti.**zeros**
Zeros of the system.

Methods

bode([w, n])	Calculate Bode magnitude and phase data of a continuous-time system.
freqresp([w, n])	Calculate the frequency response of a continuous-time system.
impulse([X0, T, N])	Return the impulse response of a continuous-time system.
output(U, T[, X0])	Return the response of a continuous-time system to input *U*.
step([X0, T, N])	Return the step response of a continuous-time system.
to_discrete(dt[, method, alpha])	Return a discretized version of the current system.

`lti.bode(w=None, n=100)`
Calculate Bode magnitude and phase data of a continuous-time system.

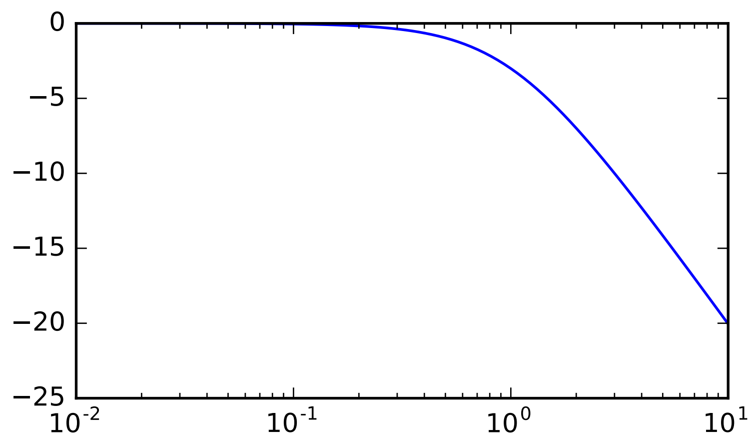
Returns a 3-tuple containing arrays of frequencies [rad/s], magnitude [dB] and phase [deg]. See `bode` for details.

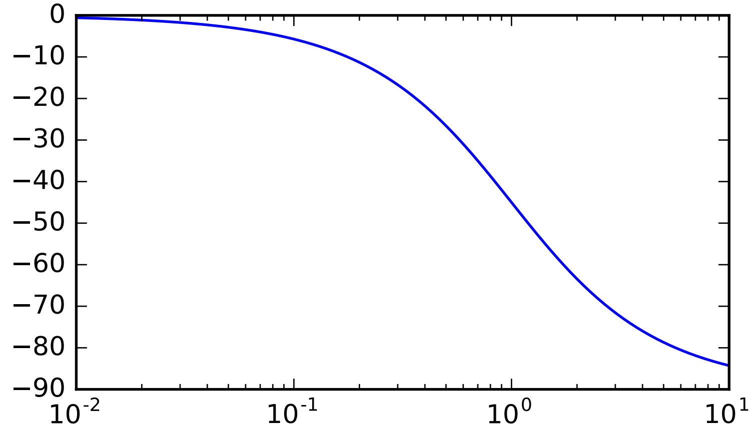
Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> sys = signal.TransferFunction([1], [1, 1])
>>> w, mag, phase = sys.bode()
```

```
>>> plt.figure()
>>> plt.semilogx(w, mag)     # Bode magnitude plot
>>> plt.figure()
>>> plt.semilogx(w, phase)   # Bode phase plot
>>> plt.show()
```





lti.**freqresp**(*w=None*, *n=10000*)
Calculate the frequency response of a continuous-time system.

Returns a 2-tuple containing arrays of frequencies [rad/s] and complex magnitude. See *freqresp* for details.

lti.**impulse**(*X0=None*, *T=None*, *N=None*)
Return the impulse response of a continuous-time system. See *impulse* for details.

lti.**output**(*U*, *T*, *X0=None*)
Return the response of a continuous-time system to input *U*. See *lsim* for details.

lti.**step**(*X0=None*, *T=None*, *N=None*)
Return the step response of a continuous-time system. See *step* for details.

lti.**to_discrete**(*dt*, *method='zoh'*, *alpha=None*)
Return a discretized version of the current system.

Parameters: See *cont2discrete* for details.

Returns sys: instance of *dlti*

class scipy.signal.**StateSpace**(**system*, ***kwargs*)
Linear Time Invariant system in state-space form.

Represents the system as the continuous-time, first order differential equation $\dot{x} = Ax + Bu$ or the discrete-time difference equation $x[k+1] = Ax[k] + Bu[k]$. *StateSpace* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

Parameters ***system: arguments**

The *StateSpace* class can be instantiated with 1 or 3 arguments. The following gives the number of input arguments and their interpretation:

- 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
- 4: array_like: (A, B, C, D)

dt: float, optional

Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, dt=0.1.

See also:

TransferFunction, *ZerosPolesGain*, *lti*, *dlti*, *ss2zpk*, *ss2tf*, *zpk2sos*

Notes

Changing the value of properties that are not part of the *StateSpace* system representation (such as *zeros* or *poles*) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the zeros, poles or gain.

Examples

```
>>> from scipy import signal
```

```
>>> a = np.array([[0, 1], [0, 0]])
>>> b = np.array([[0], [1]])
>>> c = np.array([[1, 0]])
>>> d = np.array([[0]])
```

```
>>> sys = signal.StateSpace(a, b, c, d)
>>> print(sys)
StateSpaceContinuous(
array([[0, 1],
       [0, 0]]),
array([[0],
       [1]]),
array([[1, 0]]),
array([[0]]),
dt: None
)
```

```
>>> sys.to_discrete(0.1)
StateSpaceDiscrete(
array([[ 1. ,  0.1],
       [ 0. ,  1. ]]),
array([[ 0.005],
       [ 0.1  ]]),
array([[1, 0]]),
array([[0]]),
dt: 0.1
)
```

```
>>> a = np.array([[1, 0.1], [0, 1]])
>>> b = np.array([[0.005], [0.1]])
```

```
>>> signal.StateSpace(a, b, c, d, dt=0.1)
StateSpaceDiscrete(
array([[ 1. ,  0.1],
       [ 0. ,  1. ]]),
array([[ 0.005],
       [ 0.1  ]]),
array([[1, 0]]),
array([[0]]),
dt: 0.1
)
```

Attributes

A	State matrix of the *StateSpace* system.
B	Input matrix of the *StateSpace* system.
C	Output matrix of the *StateSpace* system.
D	Feedthrough matrix of the *StateSpace* system.
den	Denominator of the *TransferFunction* system.
dt	Return the sampling time of the system, *None* for `lti` systems.
gain	Gain of the *ZerosPolesGain* system.
num	Numerator of the *TransferFunction* system.
poles	Poles of the system.
zeros	Zeros of the system.

StateSpace.**A**
State matrix of the *StateSpace* system.

StateSpace.**B**
Input matrix of the *StateSpace* system.

StateSpace.**C**
Output matrix of the *StateSpace* system.

StateSpace.**D**
Feedthrough matrix of the *StateSpace* system.

StateSpace.**den**
Denominator of the *TransferFunction* system.

StateSpace.**dt**
Return the sampling time of the system, *None* for `lti` systems.

StateSpace.**gain**
Gain of the *ZerosPolesGain* system.

StateSpace.**num**
Numerator of the *TransferFunction* system.

StateSpace.**poles**
Poles of the system.

StateSpace.**zeros**
Zeros of the system.

Methods

to_ss()	Return a copy of the current *StateSpace* system.
to_tf(**kwargs)	Convert system representation to *TransferFunction*.
to_zpk(**kwargs)	Convert system representation to *ZerosPolesGain*.

StateSpace.**to_ss**()
Return a copy of the current *StateSpace* system.

Returns **sys** : instance of *StateSpace*
The current system (copy)

StateSpace.**to_tf**(***kwargs*)
Convert system representation to *TransferFunction*.

Parameters **kwargs** : dict, optional

Additional keywords passed to *ss2zpk*

Returns **sys** : instance of *TransferFunction*

Transfer function of the current system

`StateSpace.to_zpk(**kwargs)`

Convert system representation to *ZerosPolesGain*.

Parameters **kwargs** : dict, optional

Additional keywords passed to *ss2zpk*

Returns **sys** : instance of *ZerosPolesGain*

Zeros, poles, gain representation of the current system

class `scipy.signal.TransferFunction(*system, **kwargs)`

Linear Time Invariant system class in transfer function form.

Represents the system as the continuous-time transfer function $H(s) = \sum_{i=0}^{N} b[N-i]s^i / \sum_{j=0}^{M} a[M-j]s^j$ or the discrete-time transfer function $H(s) = \sum_{i=0}^{N} b[N-i]z^i / \sum_{j=0}^{M} a[M-j]z^j$, where b are elements of the numerator *num*, a are elements of the denominator *den*, and `N == len(b) - 1`, `M == len(a) - 1`. *TransferFunction* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

Parameters ***system: arguments**

The *TransferFunction* class can be instantiated with 1 or 2 arguments. The following gives the number of input arguments and their interpretation:

- 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
- 2: array_like: (numerator, denominator)

dt: float, optional

Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, `dt=0.1`.

See also:

ZerosPolesGain, *StateSpace*, *lti*, *dlti*, *tf2ss*, *tf2zpk*, *tf2sos*

Notes

Changing the value of properties that are not part of the *TransferFunction* system representation (such as the *A*, *B*, *C*, *D* state-space matrices) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_ss()` before accessing/changing the A, B, C, D system matrices.

If (numerator, denominator) is passed in for `*system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` or `z^2 + 3z + 5` would be represented as `[1, 3, 5]`)

Examples

Construct the transfer function:

$$H(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1}$$

```
>>> from scipy import signal
```

```
>>> num = [1, 3, 3]
>>> den = [1, 2, 1]
```

```
>>> signal.TransferFunction(num, den)
TransferFunctionContinuous(
array([ 1.,  3.,  3.]),
array([ 1.,  2.,  1.]),
dt: None
)
```

Contruct the transfer function with a sampling time of 0.1 seconds:

$$H(z) = \frac{z^2 + 3z + 3}{z^2 + 2z + 1}$$

```
>>> signal.TransferFunction(num, den, dt=0.1)
TransferFunctionDiscrete(
array([ 1.,  3.,  3.]),
array([ 1.,  2.,  1.]),
dt: 0.1
)
```

Attributes

A	State matrix of the *StateSpace* system.
B	Input matrix of the *StateSpace* system.
C	Output matrix of the *StateSpace* system.
D	Feedthrough matrix of the *StateSpace* system.
den	Denominator of the *TransferFunction* system.
dt	Return the sampling time of the system, *None* for *lti* systems.
gain	Gain of the *ZerosPolesGain* system.
num	Numerator of the *TransferFunction* system.
poles	Poles of the system.
zeros	Zeros of the system.

TransferFunction.**A**
State matrix of the *StateSpace* system.

TransferFunction.**B**
Input matrix of the *StateSpace* system.

TransferFunction.**C**
Output matrix of the *StateSpace* system.

TransferFunction.**D**
Feedthrough matrix of the *StateSpace* system.

TransferFunction.**den**
Denominator of the *TransferFunction* system.

TransferFunction.**dt**
Return the sampling time of the system, *None* for *lti* systems.

TransferFunction.**gain**
Gain of the *ZerosPolesGain* system.

TransferFunction.**num**
Numerator of the *TransferFunction* system.

TransferFunction.**poles**
Poles of the system.

TransferFunction.**zeros**
Zeros of the system.

Methods

to_ss()	Convert system representation to *StateSpace*.
to_tf()	Return a copy of the current *TransferFunction* system.
to_zpk()	Convert system representation to *ZerosPolesGain*.

TransferFunction.**to_ss**()
Convert system representation to *StateSpace*.

Returns **sys** : instance of *StateSpace*
State space model of the current system

TransferFunction.**to_tf**()
Return a copy of the current *TransferFunction* system.

Returns **sys** : instance of *TransferFunction*
The current system (copy)

TransferFunction.**to_zpk**()
Convert system representation to *ZerosPolesGain*.

Returns **sys** : instance of *ZerosPolesGain*
Zeros, poles, gain representation of the current system

class scipy.signal.**ZerosPolesGain**(**system*, ***kwargs*)
Linear Time Invariant system class in zeros, poles, gain form.

Represents the system as the continuous- or discrete-time transfer function $H(s) = k \prod_i (s - z[i]) / \prod_j (s - p[j])$, where k is the *gain*, z are the *zeros* and p are the *poles*. *ZerosPolesGain* systems inherit additional functionality from the *lti*, respectively the *dlti* classes, depending on which system representation is used.

Parameters ***system** : arguments
The *ZerosPolesGain* class can be instantiated with 1 or 3 arguments. The following gives the number of input arguments and their interpretation:
- 1: *lti* or *dlti* system: (*StateSpace*, *TransferFunction* or *ZerosPolesGain*)
- 3: array_like: (zeros, poles, gain)

dt: float, optional
Sampling time [s] of the discrete-time systems. Defaults to *None* (continuous-time). Must be specified as a keyword argument, for example, dt=0.1.

See also:

TransferFunction, *StateSpace*, *lti*, *dlti*, *zpk2ss*, *zpk2tf*, *zpk2sos*

Notes

Changing the value of properties that are not part of the *ZerosPolesGain* system representation (such as the *A*, *B*, *C*, *D* state-space matrices) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_ss()` before accessing/changing the A, B, C, D system matrices.

Examples

```
>>> from scipy import signal
```

Transfer function: H(s) = 5(s - 1)(s - 2) / (s - 3)(s - 4)

```
>>> signal.ZerosPolesGain([1, 2], [3, 4], 5)
ZerosPolesGainContinuous(
array([1, 2]),
array([3, 4]),
5,
dt: None
)
```

Transfer function: H(z) = 5(z - 1)(z - 2) / (z - 3)(z - 4)

```
>>> signal.ZerosPolesGain([1, 2], [3, 4], 5, dt=0.1)
ZerosPolesGainDiscrete(
array([1, 2]),
array([3, 4]),
5,
dt: 0.1
)
```

Attributes

A	State matrix of the *StateSpace* system.
B	Input matrix of the *StateSpace* system.
C	Output matrix of the *StateSpace* system.
D	Feedthrough matrix of the *StateSpace* system.
den	Denominator of the *TransferFunction* system.
dt	Return the sampling time of the system, *None* for *lti* systems.
gain	Gain of the *ZerosPolesGain* system.
num	Numerator of the *TransferFunction* system.
poles	Poles of the *ZerosPolesGain* system.
zeros	Zeros of the *ZerosPolesGain* system.

ZerosPolesGain.**A**
State matrix of the *StateSpace* system.

ZerosPolesGain.**B**
Input matrix of the *StateSpace* system.

ZerosPolesGain.**C**
Output matrix of the *StateSpace* system.

ZerosPolesGain.**D**
Feedthrough matrix of the *StateSpace* system.

ZerosPolesGain.**den**
Denominator of the *TransferFunction* system.

ZerosPolesGain.**dt**
Return the sampling time of the system, *None* for *lti* systems.

ZerosPolesGain.**gain**
Gain of the *ZerosPolesGain* system.

ZerosPolesGain.**num**
Numerator of the *TransferFunction* system.

ZerosPolesGain.**poles**
Poles of the *ZerosPolesGain* system.

ZerosPolesGain.**zeros**
Zeros of the *ZerosPolesGain* system.

Methods

to_ss()	Convert system representation to *StateSpace*.
to_tf()	Convert system representation to *TransferFunction*.
to_zpk()	Return a copy of the current 'ZerosPolesGain' system.

ZerosPolesGain.**to_ss**()
Convert system representation to *StateSpace*.

Returns **sys** : instance of *StateSpace*
State space model of the current system

ZerosPolesGain.**to_tf**()
Convert system representation to *TransferFunction*.

Returns **sys** : instance of *TransferFunction*
Transfer function of the current system

ZerosPolesGain.**to_zpk**()
Return a copy of the current 'ZerosPolesGain' system.

Returns **sys** : instance of *ZerosPolesGain*
The current system (copy)

scipy.signal.**lsim**(*system*, *U*, *T*, *X0=None*, *interp=True*)
Simulate output of a continuous-time linear system.

Parameters **system** : an instance of the LTI class or a tuple describing the system.
The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of *lti*)
- 2: (num, den)
- 3: (zeros, poles, gain)
- 4: (A, B, C, D)

U : array_like
An input array describing the input at each time *T* (interpolation is assumed between given times). If there are multiple inputs, then each column of the rank-2 array represents an input. If U = 0 or None, a zero input is used.

T : array_like
The time steps at which the input is defined and at which the output is desired. Must be nonnegative, increasing, and equally spaced.

X0 : array_like, optional
The initial conditions on the state vector (zero by default).

interp : bool, optional
Whether to use linear (True, the default) or zero-order-hold (False) interpolation for the input array.

Returns **T** : 1D ndarray

Time values for the output.

yout : 1D ndarray

System response.

xout : ndarray

Time evolution of the state vector.

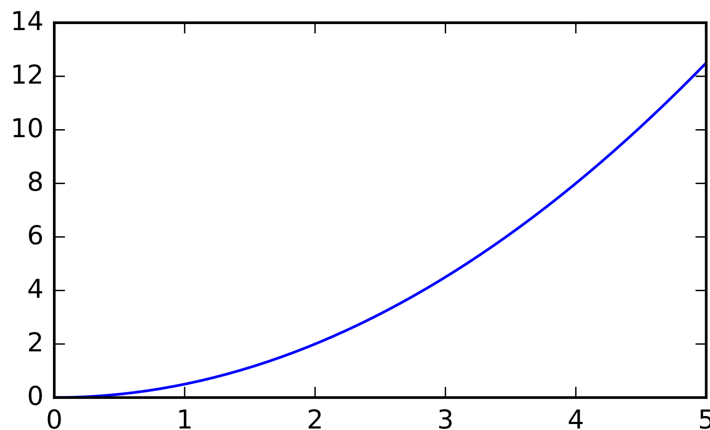
Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

Examples

Simulate a double integrator y'' = u, with a constant input u = 1

```
>>> from scipy import signal
>>> system = signal.lti([[0., 1.], [0., 0.]], [[0.], [1.]], [[1., 0.]], 0.)
>>> t = np.linspace(0, 5)
>>> u = np.ones_like(t)
>>> tout, y, x = signal.lsim(system, u, t)
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, y)
```



`scipy.signal.`**`lsim2`**(*system*, *U=None*, *T=None*, *X0=None*, ***kwargs*)

Simulate output of a continuous-time linear system, by using the ODE solver *`scipy.integrate.odeint`*.

Parameters **system** : an instance of the *`lti`* class or a tuple describing the system.

The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of *`lti`*)
- 2: (num, den)
- 3: (zeros, poles, gain)
- 4: (A, B, C, D)

U : array_like (1D or 2D), optional

An input array describing the input at each time T. Linear interpolation is used between given times. If there are multiple inputs, then each column of the rank-2 array represents an input. If U is not given, the input is assumed to be zero.

T : array_like (1D or 2D), optional

The time steps at which the input is defined and at which the output is desired. The default is 101 evenly spaced points on the interval [0,10.0].

X0 : array_like (1D), optional

The initial condition of the state vector. If *X0* is not given, the initial conditions are assumed to be 0.

kwargs : dict

Additional keyword arguments are passed on to the function *odeint*. See the notes below for more details.

Returns **T** : 1D ndarray

The time values for the output.

yout : ndarray

The response of the system.

xout : ndarray

The time-evolution of the state-vector.

Notes

This function uses `scipy.integrate.odeint` to solve the system's differential equations. Additional keyword arguments given to `lsim2` are passed on to *odeint*. See the documentation for `scipy.integrate.odeint` for the full list of arguments.

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

`scipy.signal.`**`impulse`**(*system*, *X0=None*, *T=None*, *N=None*)

Impulse response of continuous-time system.

Parameters **system** : an instance of the LTI class or a tuple of array_like

describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)
- 4 (A, B, C, D)

X0 : array_like, optional

Initial state-vector. Defaults to zero.

T : array_like, optional

Time points. Computed if not given.

N : int, optional

The number of time points to compute (if *T* is not given).

Returns **T** : ndarray

A 1-D array of time points.

yout : ndarray

A 1-D array containing the impulse response of the system (except for singularities at zero).

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

`scipy.signal.`**`impulse2`**(*system*, *X0=None*, *T=None*, *N=None*, ***kwargs*)

Impulse response of a single-input, continuous-time linear system.

Parameters **system** : an instance of the LTI class or a tuple of array_like

describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)
- 4 (A, B, C, D)

X0 : 1-D array_like, optional

The initial condition of the state vector. Default: 0 (the zero vector).

T : 1-D array_like, optional

The time steps at which the input is defined and at which the output is desired. If *T* is not given, the function will generate a set of time samples automatically.

N : int, optional

Number of time points to compute. Default: 100.

kwargs : various types

Additional keyword arguments are passed on to the function `scipy.signal.lsim2`, which in turn passes them on to `scipy.integrate.odeint`; see the latter's documentation for information about these arguments.

Returns **T** : ndarray

The time values for the output.

yout : ndarray

The output response of the system.

See also:

`impulse`, `lsim2`, `integrate.odeint`

Notes

The solution is generated by calling `scipy.signal.lsim2`, which uses the differential equation solver `scipy.integrate.odeint`.

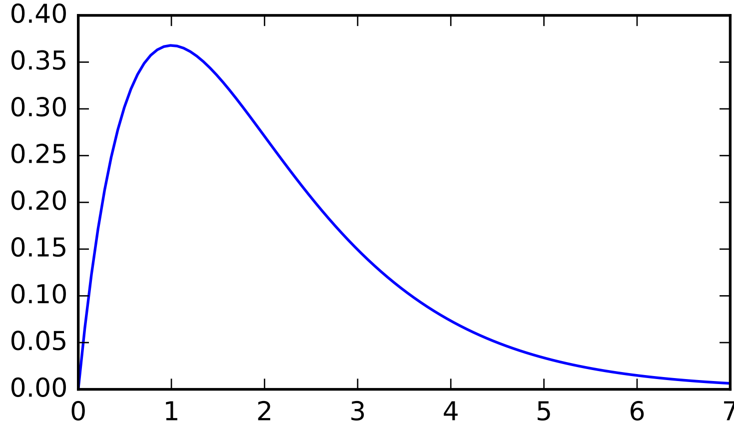
If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

New in version 0.8.0.

Examples

Second order system with a repeated root: x''(t) + 2*x(t) + x(t) = u(t)

```
>>> from scipy import signal
>>> system = ([1.0], [1.0, 2.0, 1.0])
>>> t, y = signal.impulse2(system)
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, y)
```



`scipy.signal.step(system, X0=None, T=None, N=None)`

Step response of continuous-time system.

Parameters **system** : an instance of the LTI class or a tuple of array_like

describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)
- 4 (A, B, C, D)

X0 : array_like, optional

Initial state-vector (default is zero).

T : array_like, optional

Time points (computed if not given).

N : int, optional

Number of time points to compute if *T* is not given.

Returns **T** : 1D ndarray

Output time points.

yout : 1D ndarray

Step response of system.

See also:

scipy.signal.step2

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

`scipy.signal.step2(system, X0=None, T=None, N=None, **kwargs)`

Step response of continuous-time system.

This function is functionally the same as *scipy.signal.step*, but it uses the function *scipy.signal.lsim2* to compute the step response.

Parameters **system** : an instance of the LTI class or a tuple of array_like

describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)

- 4 (A, B, C, D)

X0 : array_like, optional
Initial state-vector (default is zero).

T : array_like, optional
Time points (computed if not given).

N : int, optional
Number of time points to compute if *T* is not given.

kwargs : various types
Additional keyword arguments are passed on the function `scipy.signal.lsim2`, which in turn passes them on to `scipy.integrate.odeint`. See the documentation for `scipy.integrate.odeint` for information about these arguments.

Returns

T : 1D ndarray
Output time points.

yout : 1D ndarray
Step response of system.

See also:

`scipy.signal.step`

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

New in version 0.8.0.

`scipy.signal.`**`freqresp`**`(`*system, w=None, n=10000*`)`

Calculate the frequency response of a continuous-time system.

Parameters

system : an instance of the `lti` class or a tuple describing the system.
The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)
- 4 (A, B, C, D)

w : array_like, optional
Array of frequencies (in rad/s). Magnitude and phase data is calculated for every value in this array. If not given, a reasonable set will be calculated.

n : int, optional
Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

Returns

w : 1D ndarray
Frequency array [rad/s]

H : 1D ndarray
Array of complex magnitude values

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

Examples

Generating the Nyquist plot of a transfer function

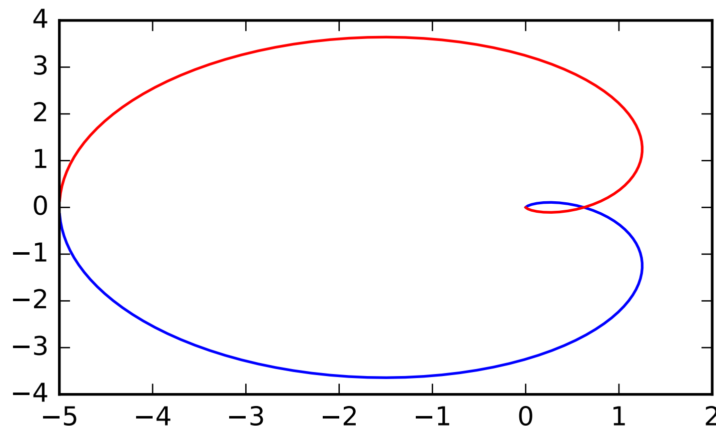
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Transfer function: H(s) = 5 / (s-1)^3

```
>>> s1 = signal.ZerosPolesGain([], [1, 1, 1], [5])
```

```
>>> w, H = signal.freqresp(s1)
```

```
>>> plt.figure()
>>> plt.plot(H.real, H.imag, "b")
>>> plt.plot(H.real, -H.imag, "r")
>>> plt.show()
```



`scipy.signal.bode(system, w=None, n=100)`

Calculate Bode magnitude and phase data of a continuous-time system.

Parameters **system** : an instance of the LTI class or a tuple describing the system.

The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `lti`)
- 2 (num, den)
- 3 (zeros, poles, gain)
- 4 (A, B, C, D)

w : array_like, optional

Array of frequencies (in rad/s). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.

n : int, optional

Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

Returns **w** : 1D ndarray

Frequency array [rad/s]

mag : 1D ndarray

Magnitude array [dB]

phase : 1D ndarray

Phase array [deg]

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `s^2 + 3s + 5` would be represented as `[1, 3, 5]`).

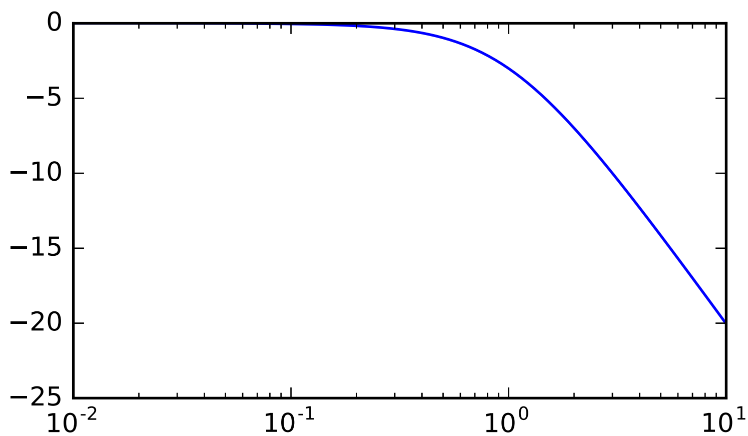
New in version 0.11.0.

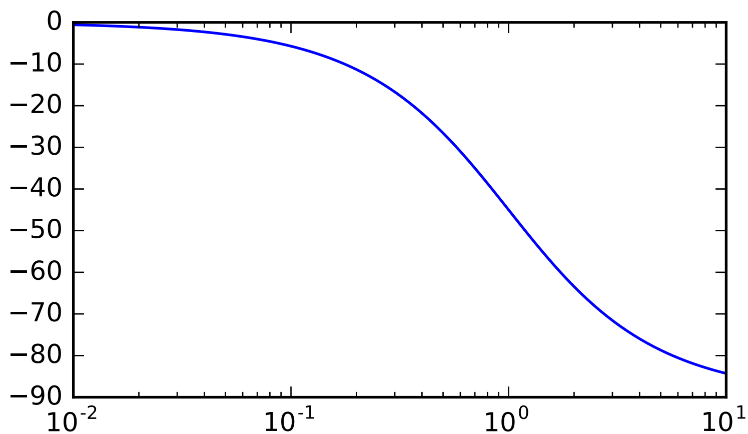
Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> sys = signal.TransferFunction([1], [1, 1])
>>> w, mag, phase = signal.bode(sys)
```

```
>>> plt.figure()
>>> plt.semilogx(w, mag)    # Bode magnitude plot
>>> plt.figure()
>>> plt.semilogx(w, phase)  # Bode phase plot
>>> plt.show()
```





5.20.7 Discrete-Time Linear Systems

dlti(*system, **kwargs)	Discrete-time linear time invariant system base class.
StateSpace(*system, **kwargs)	Linear Time Invariant system in state-space form.
TransferFunction(*system, **kwargs)	Linear Time Invariant system class in transfer function form.
ZerosPolesGain(*system, **kwargs)	Linear Time Invariant system class in zeros, poles, gain form.
dlsim(system, u[, t, x0])	Simulate output of a discrete-time linear system.
dimpulse(system[, x0, t, n])	Impulse response of discrete-time system.
dstep(system[, x0, t, n])	Step response of discrete-time system.
dfreqresp(system[, w, n, whole])	Calculate the frequency response of a discrete-time system.
dbode(system[, w, n])	Calculate Bode magnitude and phase data of a discrete-time system.

class `scipy.signal.`**`dlti`**(**system*, ***kwargs*)

Discrete-time linear time invariant system base class.

Parameters ***system: arguments**

The *dlti* class can be instantiated with either 2, 3 or 4 arguments. The following gives the number of arguments and the corresponding discrete-time subclass that is created:

- 2: *TransferFunction*: (numerator, denominator)
- 3: *ZerosPolesGain*: (zeros, poles, gain)
- 4: *StateSpace*: (A, B, C, D)

Each argument can be an array or a sequence.

dt: float, optional

Sampling time [s] of the discrete-time systems. Defaults to `True` (unspecified sampling time). Must be specified as a keyword argument, for example, `dt=0.1`.

See also:

ZerosPolesGain, *StateSpace*, *TransferFunction*, *lti*

Notes

dlti instances do not exist directly. Instead, *dlti* creates an instance of one of its subclasses: *StateSpace*, *TransferFunction* or *ZerosPolesGain*.

Changing the value of properties that are not directly part of the current system representation (such as the *zeros* of a *StateSpace* system) is very inefficient and may lead to numerical inaccuracies. It is better to convert to the specific system representation first. For example, call `sys = sys.to_zpk()` before accessing/changing the zeros, poles or gain.

If (numerator, denominator) is passed in for `*system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g., `z^2 + 3z + 5` would be represented as `[1, 3, 5]`).

New in version 0.18.0.

Examples

```
>>> from scipy import signal
```

```
>>> signal.dlti(1, 2, 3, 4)
StateSpaceDiscrete(
```

```
array([[1]]),
array([[2]]),
array([[3]]),
array([[4]]),
dt: True
)
```

```
>>> signal.dlti(1, 2, 3, 4, dt=0.1)
StateSpaceDiscrete(
array([[1]]),
array([[2]]),
array([[3]]),
array([[4]]),
dt: 0.1
)
```

```
>>> signal.dlti([1, 2], [3, 4], 5, dt=0.1)
ZerosPolesGainDiscrete(
array([1, 2]),
array([3, 4]),
5,
dt: 0.1
)
```

```
>>> signal.dlti([3, 4], [1, 2], dt=0.1)
TransferFunctionDiscrete(
array([ 3.,  4.]),
array([ 1.,  2.]),
dt: 0.1
)
```

Attributes

A	State matrix of the *StateSpace* system.
B	Input matrix of the *StateSpace* system.
C	Output matrix of the *StateSpace* system.
D	Feedthrough matrix of the *StateSpace* system.
den	Denominator of the *TransferFunction* system.
dt	Return the sampling time of the system.
gain	Gain of the *ZerosPolesGain* system.
num	Numerator of the *TransferFunction* system.
poles	Poles of the system.
zeros	Zeros of the system.

dlti.**A**

State matrix of the *StateSpace* system.

dlti.**B**

Input matrix of the *StateSpace* system.

dlti.**C**

Output matrix of the *StateSpace* system.

dlti.**D**

Feedthrough matrix of the *StateSpace* system.

`dlti.`**`den`**
Denominator of the *TransferFunction* system.

`dlti.`**`dt`**
Return the sampling time of the system.

`dlti.`**`gain`**
Gain of the *ZerosPolesGain* system.

`dlti.`**`num`**
Numerator of the *TransferFunction* system.

`dlti.`**`poles`**
Poles of the system.

`dlti.`**`zeros`**
Zeros of the system.

Methods

bode([w, n])	Calculate Bode magnitude and phase data of a discrete-time system.
freqresp([w, n, whole])	Calculate the frequency response of a discrete-time system.
impulse([x0, t, n])	Return the impulse response of the discrete-time *dlti* system.
output(u, t[, x0])	Return the response of the discrete-time system to input *u*.
step([x0, t, n])	Return the step response of the discrete-time *dlti* system.

`dlti.`**`bode`**(*w=None*, *n=100*)
Calculate Bode magnitude and phase data of a discrete-time system.

Returns a 3-tuple containing arrays of frequencies [rad/s], magnitude [dB] and phase [deg]. See *dbode* for details.

Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

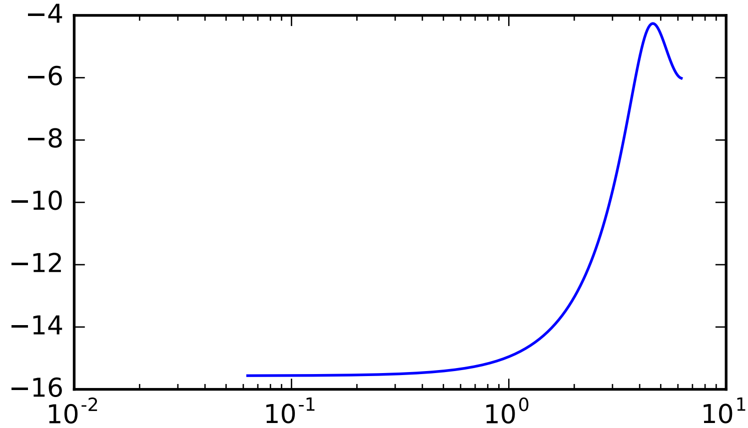
Transfer function: H(z) = 1 / (z^2 + 2z + 3) with sampling time 0.5s

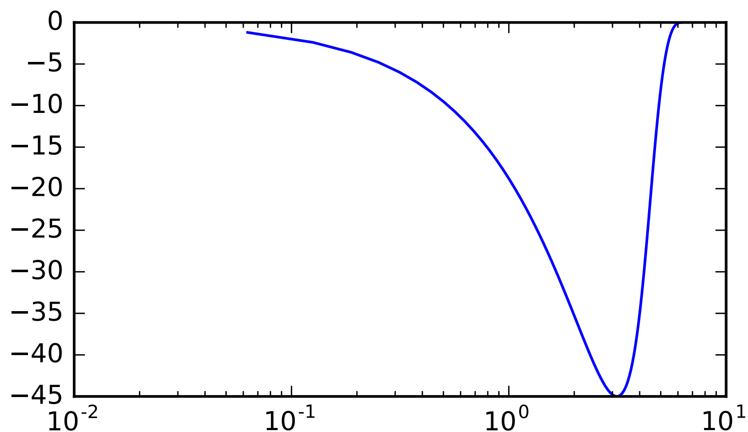
```
>>> sys = signal.TransferFunction([1], [1, 2, 3], dt=0.5)
```

Equivalent: signal.dbode(sys)

```
>>> w, mag, phase = sys.bode()
```

```
>>> plt.figure()
>>> plt.semilogx(w, mag)    # Bode magnitude plot
>>> plt.figure()
>>> plt.semilogx(w, phase)  # Bode phase plot
>>> plt.show()
```





`dlti.`**`freqresp`**(*w=None*, *n=10000*, *whole=False*)
Calculate the frequency response of a discrete-time system.

Returns a 2-tuple containing arrays of frequencies [rad/s] and complex magnitude. See *`dfreqresp`* for details.

`dlti.`**`impulse`**(*x0=None*, *t=None*, *n=None*)
Return the impulse response of the discrete-time *`dlti`* system. See *`dimpulse`* for details.

`dlti.`**`output`**(*u*, *t*, *x0=None*)
Return the response of the discrete-time system to input *u*. See *`dlsim`* for details.

`dlti.`**`step`**(*x0=None*, *t=None*, *n=None*)
Return the step response of the discrete-time *`dlti`* system. See *`dstep`* for details.

`scipy.signal.`**`dlsim`**(*system*, *u*, *t=None*, *x0=None*)
Simulate output of a discrete-time linear system.

Parameters **system** : tuple of array_like or instance of *`dlti`*

A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of *dlti*)
- 3: (num, den, dt)
- 4: (zeros, poles, gain, dt)
- 5: (A, B, C, D, dt)

u : array_like
An input array describing the input at each time *t* (interpolation is assumed between given times). If there are multiple inputs, then each column of the rank-2 array represents an input.

t : array_like, optional
The time steps at which the input is defined. If *t* is given, it must be the same length as *u*, and the final value in *t* determines the number of steps returned in the output.

x0 : array_like, optional
The initial conditions on the state vector (zero by default).

Returns

tout : ndarray
Time values for the output, as a 1-D array.

yout : ndarray
System response, as a 1-D array.

xout : ndarray, optional
Time-evolution of the state-vector. Only generated if the input is a *StateSpace* system.

See also:

lsim, *dstep*, *dimpulse*, *cont2discrete*

Examples

A simple integrator transfer function with a discrete time step of 1.0 could be implemented as:

```
>>> from scipy import signal
>>> tf = ([1.0,], [1.0, -1.0], 1.0)
>>> t_in = [0.0, 1.0, 2.0, 3.0]
>>> u = np.asarray([0.0, 0.0, 1.0, 1.0])
>>> t_out, y = signal.dlsim(tf, u, t=t_in)
>>> y.T
array([[ 0.,  0.,  0.,  1.]])
```

scipy.signal.**dimpulse**(*system*, *x0=None*, *t=None*, *n=None*)

Impulse response of discrete-time system.

Parameters

system : tuple of array_like or instance of *dlti*
A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of *dlti*)
- 3: (num, den, dt)
- 4: (zeros, poles, gain, dt)
- 5: (A, B, C, D, dt)

x0 : array_like, optional
Initial state-vector. Defaults to zero.

t : array_like, optional
Time points. Computed if not given.

n : int, optional
The number of time points to compute (if *t* is not given).

Returns

tout : ndarray
Time values for the output, as a 1-D array.

yout : ndarray
Impulse response of system. Each element of the tuple represents the output of the system based on an impulse in each input.

See also:

impulse, *dstep*, *dlsim*, *cont2discrete*

`scipy.signal.dstep(system, x0=None, t=None, n=None)`

Step response of discrete-time system.

Parameters **system** : tuple of array_like

A tuple describing the system. The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of `dlti`)
- 3: (num, den, dt)
- 4: (zeros, poles, gain, dt)
- 5: (A, B, C, D, dt)

x0 : array_like, optional

Initial state-vector. Defaults to zero.

t : array_like, optional

Time points. Computed if not given.

n : int, optional

The number of time points to compute (if *t* is not given).

Returns **tout** : ndarray

Output time points, as a 1-D array.

yout : ndarray

Step response of system. Each element of the tuple represents the output of the system based on a step response to each input.

See also:

step, *dimpulse*, *dlsim*, *cont2discrete*

`scipy.signal.dfreqresp(system, w=None, n=10000, whole=False)`

Calculate the frequency response of a discrete-time system.

Parameters **system** : an instance of the `dlti` class or a tuple describing the system.

The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `dlti`)
- 2 (numerator, denominator, dt)
- 3 (zeros, poles, gain, dt)
- 4 (A, B, C, D, dt)

w : array_like, optional

Array of frequencies (in radians/sample). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.

n : int, optional

Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

whole : bool, optional

Normally, if 'w' is not given, frequencies are computed from 0 to the Nyquist frequency, pi radians/sample (upper-half of unit-circle). If *whole* is True, compute frequencies from 0 to 2*pi radians/sample.

Returns **w** : 1D ndarray

Frequency array [radians/sample]

H : 1D ndarray

Array of complex magnitude values

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `z^2 + 3z + 5` would be represented as `[1, 3, 5]`).

New in version 0.18.0.

Examples

Generating the Nyquist plot of a transfer function

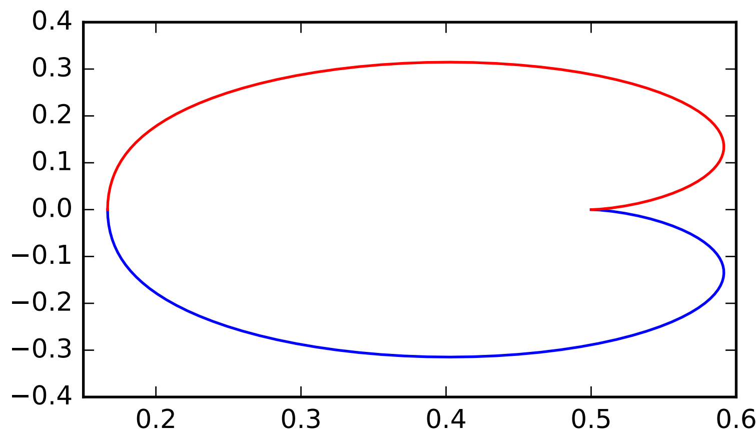
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Transfer function: H(z) = 1 / (z^2 + 2z + 3)

```
>>> sys = signal.TransferFunction([1], [1, 2, 3], dt=0.05)
```

```
>>> w, H = signal.dfreqresp(sys)
```

```
>>> plt.figure()
>>> plt.plot(H.real, H.imag, "b")
>>> plt.plot(H.real, -H.imag, "r")
>>> plt.show()
```



`scipy.signal.dbode(system, w=None, n=100)`

Calculate Bode magnitude and phase data of a discrete-time system.

Parameters

system : an instance of the LTI class or a tuple describing the system.
The following gives the number of elements in the tuple and the interpretation:

- 1 (instance of `dlti`)
- 2 (num, den, dt)
- 3 (zeros, poles, gain, dt)
- 4 (A, B, C, D, dt)

w : array_like, optional
Array of frequencies (in radians/sample). Magnitude and phase data is calculated for every value in this array. If not given a reasonable set will be calculated.

n : int, optional
Number of frequency points to compute if *w* is not given. The *n* frequencies are logarithmically spaced in an interval chosen to include the influence of the poles and zeros of the system.

Returns

w : 1D ndarray

Frequency array [rad/time_unit]

mag : 1D ndarray

Magnitude array [dB]

phase : 1D ndarray

Phase array [deg]

Notes

If (num, den) is passed in for `system`, coefficients for both the numerator and denominator should be specified in descending exponent order (e.g. `z^2 + 3z + 5` would be represented as `[1, 3, 5]`).

New in version 0.18.0.

Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

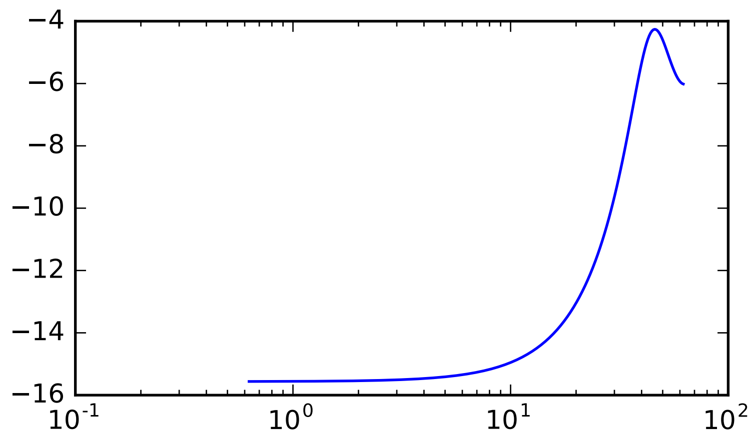
Transfer function: H(z) = 1 / (z^2 + 2z + 3)

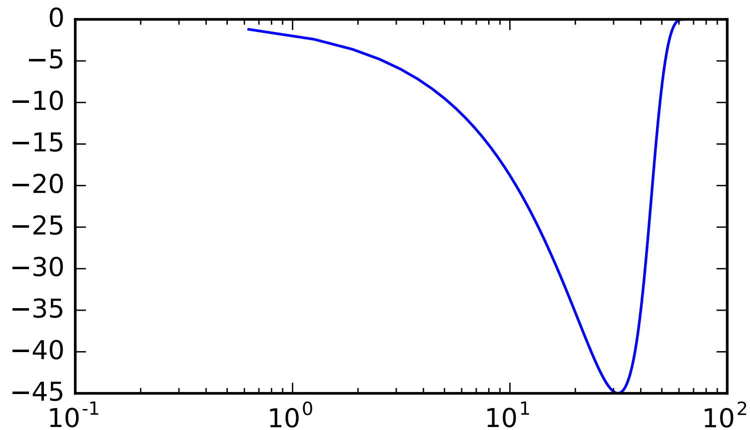
```
>>> sys = signal.TransferFunction([1], [1, 2, 3], dt=0.05)
```

Equivalent: sys.bode()

```
>>> w, mag, phase = signal.dbode(sys)
```

```
>>> plt.figure()
>>> plt.semilogx(w, mag)    # Bode magnitude plot
>>> plt.figure()
>>> plt.semilogx(w, phase)  # Bode phase plot
>>> plt.show()
```





5.20.8 LTI Representations

tf2zpk(b, a)	Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.
tf2sos(b, a[, pairing])	Return second-order sections from transfer function representation
tf2ss(num, den)	Transfer function to state-space representation.
zpk2tf(z, p, k)	Return polynomial transfer function representation from zeros and poles
zpk2sos(z, p, k[, pairing])	Return second-order sections from zeros, poles, and gain of a system
zpk2ss(z, p, k)	Zero-pole-gain representation to state-space representation
ss2tf(A, B, C, D[, input])	State-space to transfer function.
ss2zpk(A, B, C, D[, input])	State-space representation to zero-pole-gain representation.
sos2zpk(sos)	Return zeros, poles, and gain of a series of second-order sections
sos2tf(sos)	Return a single transfer function from a series of second-order sections
cont2discrete(system, dt[, method, alpha])	Transform a continuous to a discrete state-space system.
place_poles(A, B, poles[, method, rtol, maxiter])	Compute K such that eigenvalues (A - dot(B, K))=poles.

`scipy.signal.`**`tf2zpk`**(*b*, *a*)

Return zero, pole, gain (z, p, k) representation from a numerator, denominator representation of a linear filter.

Parameters

b : array_like

Numerator polynomial coefficients.

a : array_like

Denominator polynomial coefficients.

Returns

z : ndarray

Zeros of the transfer function.

p : ndarray

Poles of the transfer function.

k : float
System gain.

Notes

If some values of *b* are too close to 0, they are removed. In that case, a BadCoefficients warning is emitted.

The *b* and *a* arrays are interpreted as coefficients for positive, descending powers of the transfer function variable. So the inputs $b = [b_0, b_1, ..., b_M]$ and $a = [a_0, a_1, ..., a_N]$ can represent an analog filter of the form:

$$H(s) = \frac{b_0 s^M + b_1 s^{(M-1)} + \cdots + b_M}{a_0 s^N + a_1 s^{(N-1)} + \cdots + a_N}$$

or a discrete-time filter of the form:

$$H(z) = \frac{b_0 z^M + b_1 z^{(M-1)} + \cdots + b_M}{a_0 z^N + a_1 z^{(N-1)} + \cdots + a_N}$$

This "positive powers" form is found more commonly in controls engineering. If *M* and *N* are equal (which is true for all filters generated by the bilinear transform), then this happens to be equivalent to the "negative powers" discrete-time form preferred in DSP:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \cdots + a_N z^{-N}}$$

Although this is true for common filters, remember that this is not true in the general case. If *M* and *N* are not equal, the discrete-time transfer function coefficients must first be converted to the "positive powers" form before finding the poles and zeros.

`scipy.signal.`**`tf2sos`**`(`*b*, *a*, *pairing='nearest'*`)`

Return second-order sections from transfer function representation

Parameters

b : array_like
Numerator polynomial coefficients.

a : array_like
Denominator polynomial coefficients.

pairing : {'nearest', 'keep_odd'}, optional
The method to use to combine pairs of poles and zeros into sections. See *zpk2sos*.

Returns

sos : ndarray
Array of second-order filter coefficients, with shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

See also:

zpk2sos, *sosfilt*

Notes

It is generally discouraged to convert from TF to SOS format, since doing so usually will not improve numerical precision errors. Instead, consider designing filters in ZPK format and converting directly to SOS. TF is converted to SOS by first converting to ZPK format, then converting ZPK to SOS.

New in version 0.16.0.

`scipy.signal.`**`tf2ss`**`(`*num*, *den*`)`

Transfer function to state-space representation.

Parameters

num, den : array_like
Sequences representing the coefficients of the numerator and denominator polynomials, in order of descending degree. The denominator needs to be at least as long as the numerator.

Returns

A, B, C, D : ndarray
State space representation of the system, in controller canonical form.

Examples

Convert the transfer function:

$$H(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1}$$

```
>>> num = [1, 3, 3]
>>> den = [1, 2, 1]
```

to the state-space representation:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = \begin{bmatrix} 1 & 2 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \end{bmatrix} \mathbf{u}(t)$$

```
>>> from scipy.signal import tf2ss
>>> A, B, C, D = tf2ss(num, den)
>>> A
array([[-2., -1.],
       [ 1.,  0.]])
>>> B
array([[ 1.],
       [ 0.]])
>>> C
array([[ 1.,  2.]])
>>> D
array([[ 1.]])
```

`scipy.signal.`**`zpk2tf`**(*z*, *p*, *k*)

Return polynomial transfer function representation from zeros and poles

Parameters **z** : array_like

Zeros of the transfer function.

p : array_like

Poles of the transfer function.

k : float

System gain.

Returns **b** : ndarray

Numerator polynomial coefficients.

a : ndarray

Denominator polynomial coefficients.

`scipy.signal.`**`zpk2sos`**(*z*, *p*, *k*, *pairing='nearest'*)

Return second-order sections from zeros, poles, and gain of a system

Parameters **z** : array_like

Zeros of the transfer function.

p : array_like

Poles of the transfer function.

k : float

System gain.

pairing : {'nearest', 'keep_odd'}, optional

The method to use to combine pairs of poles and zeros into sections. See Notes below.

Returns **sos** : ndarray

Array of second-order filter coefficients, with shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

See also:

sosfilt

Notes

The algorithm used to convert ZPK to SOS format is designed to minimize errors due to numerical precision issues. The pairing algorithm attempts to minimize the peak gain of each biquadratic section. This is done by pairing poles with the nearest zeros, starting with the poles closest to the unit circle.

Algorithms

The current algorithms are designed specifically for use with digital filters. (The output coefficents are not correct for analog filters.)

The steps in the `pairing='nearest'` and `pairing='keep_odd'` algorithms are mostly shared. The `nearest` algorithm attempts to minimize the peak gain, while `'keep_odd'` minimizes peak gain under the constraint that odd-order systems should retain one section as first order. The algorithm steps and are as follows:

As a pre-processing step, add poles or zeros to the origin as necessary to obtain the same number of poles and zeros for pairing. If `pairing == 'nearest'` and there are an odd number of poles, add an additional pole and a zero at the origin.

The following steps are then iterated over until no more poles or zeros remain:

1. Take the (next remaining) pole (complex or real) closest to the unit circle to begin a new filter section.
2. If the pole is real and there are no other remaining real poles[1], add the closest real zero to the section and leave it as a first order section. Note that after this step we are guaranteed to be left with an even number of real poles, complex poles, real zeros, and complex zeros for subsequent pairing iterations.
3. Else:
   (a) If the pole is complex and the zero is the only remaining real zero*, then pair the pole with the *next* closest zero (guaranteed to be complex). This is necessary to ensure that there will be a real zero remaining to eventually create a first-order section (thus keeping the odd order).
   (b) Else pair the pole with the closest remaining zero (complex or real).
   (c) Proceed to complete the second-order section by adding another pole and zero to the current pole and zero in the section:
      i. If the current pole and zero are both complex, add their conjugates.
      ii. Else if the pole is complex and the zero is real, add the conjugate pole and the next closest real zero.
      iii. Else if the pole is real and the zero is complex, add the conjugate zero and the real pole closest to those zeros.
      iv. Else (we must have a real pole and real zero) add the next real pole closest to the unit circle, and then add the real zero closest to that pole.

New in version 0.16.0.

Examples

Design a 6th order low-pass elliptic digital filter for a system with a sampling rate of 8000 Hz that has a pass-band corner frequency of 1000 Hz. The ripple in the pass-band should not exceed 0.087 dB, and the attenuation in the stop-band should be at least 90 dB.

In the following call to *signal.ellip*, we could use `output='sos'`, but for this example, we'll use `output='zpk'`, and then convert to SOS format with *zpk2sos*:

```
>>> from scipy import signal
>>> z, p, k = signal.ellip(6, 0.087, 90, 1000/(0.5*8000), output='zpk')
```

[1] This conditional can only be met for specific odd-order inputs with the `pairing == 'keep_odd'` method.

Now convert to SOS format.

```
>>> sos = signal.zpk2sos(z, p, k)
```

The coefficients of the numerators of the sections:

```
>>> sos[:, :3]
array([[ 0.0014154 ,  0.00248707,  0.0014154 ],
       [ 1.        ,  0.72965193,  1.        ],
       [ 1.        ,  0.17594966,  1.        ]])
```

The symmetry in the coefficients occurs because all the zeros are on the unit circle.

The coefficients of the denominators of the sections:

```
>>> sos[:, 3:]
array([[ 1.        , -1.32543251,  0.46989499],
       [ 1.        , -1.26117915,  0.6262586 ],
       [ 1.        , -1.25707217,  0.86199667]])
```

The next example shows the effect of the *pairing* option. We have a system with three poles and three zeros, so the SOS array will have shape (2, 6). The means there is, in effect, an extra pole and an extra zero at the origin in the SOS representation.

```
>>> z1 = np.array([-1, -0.5-0.5j, -0.5+0.5j])
>>> p1 = np.array([0.75, 0.8+0.1j, 0.8-0.1j])
```

With `pairing='nearest'` (the default), we obtain

```
>>> signal.zpk2sos(z1, p1, 1)
array([[ 1.  ,  1.  ,  0.5 ,  1.  , -0.75,  0.  ],
       [ 1.  ,  1.  ,  0.  ,  1.  , -1.6 ,  0.65]])
```

The first section has the zeros {-0.5-0.05j, -0.5+0.5j} and the poles {0, 0.75}, and the second section has the zeros {-1, 0} and poles {0.8+0.1j, 0.8-0.1j}. Note that the extra pole and zero at the origin have been assigned to different sections.

With `pairing='keep_odd'`, we obtain:

```
>>> signal.zpk2sos(z1, p1, 1, pairing='keep_odd')
array([[ 1.  ,  1.  ,  0.  ,  1.  , -0.75,  0.  ],
       [ 1.  ,  1.  ,  0.5 ,  1.  , -1.6 ,  0.65]])
```

The extra pole and zero at the origin are in the same section. The first section is, in effect, a first-order section.

`scipy.signal.`**`zpk2ss`**(*z*, *p*, *k*)

Zero-pole-gain representation to state-space representation

Parameters **z, p** : sequence

Zeros and poles.

k : float

System gain.

Returns **A, B, C, D** : ndarray

State space representation of the system, in controller canonical form.

`scipy.signal.`**`ss2tf`**(*A*, *B*, *C*, *D*, *input=0*)

State-space to transfer function.

A, B, C, D defines a linear state-space system with *p* inputs, *q* outputs, and *n* state variables.

Parameters **A** : array_like

State (or system) matrix of shape `(n, n)`

B : array_like
Input matrix of shape `(n, p)`

C : array_like
Output matrix of shape `(q, n)`

D : array_like
Feedthrough (or feedforward) matrix of shape `(q, p)`

input : int, optional
For multiple-input systems, the index of the input to use.

Returns **num** : 2-D ndarray
Numerator(s) of the resulting transfer function(s). *num* has one row for each of the system's outputs. Each row is a sequence representation of the numerator polynomial.

den : 1-D ndarray
Denominator of the resulting transfer function(s). *den* is a sequence representation of the denominator polynomial.

Examples

Convert the state-space representation:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = \begin{bmatrix} 1 & 2 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \end{bmatrix} \mathbf{u}(t)$$

```
>>> A = [[-2, -1], [1, 0]]
>>> B = [[1], [0]]  # 2-dimensional column vector
>>> C = [[1, 2]]    # 2-dimensional row vector
>>> D = 1
```

to the transfer function:

$$H(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1}$$

```
>>> from scipy.signal import ss2tf
>>> ss2tf(A, B, C, D)
(array([[1, 3, 3]]), array([ 1.,  2.,  1.]))
```

`scipy.signal.`**`ss2zpk`**(*A*, *B*, *C*, *D*, *input=0*)
State-space representation to zero-pole-gain representation.

A, B, C, D defines a linear state-space system with *p* inputs, *q* outputs, and *n* state variables.

Parameters **A** : array_like
State (or system) matrix of shape `(n, n)`

B : array_like
Input matrix of shape `(n, p)`

C : array_like
Output matrix of shape `(q, n)`

D : array_like
Feedthrough (or feedforward) matrix of shape `(q, p)`

input : int, optional
For multiple-input systems, the index of the input to use.

Returns **z, p** : sequence

Zeros and poles.

k : float

System gain.

`scipy.signal.sos2zpk(sos)`

Return zeros, poles, and gain of a series of second-order sections

Parameters **sos** : array_like

Array of second-order filter coefficients, must have shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

Returns **z** : ndarray

Zeros of the transfer function.

p : ndarray

Poles of the transfer function.

k : float

System gain.

Notes

New in version 0.16.0.

`scipy.signal.sos2tf(sos)`

Return a single transfer function from a series of second-order sections

Parameters **sos** : array_like

Array of second-order filter coefficients, must have shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

Returns **b** : ndarray

Numerator polynomial coefficients.

a : ndarray

Denominator polynomial coefficients.

Notes

New in version 0.16.0.

`scipy.signal.cont2discrete(system, dt, method='zoh', alpha=None)`

Transform a continuous to a discrete state-space system.

Parameters **system** : a tuple describing the system or an instance of *lti*

The following gives the number of elements in the tuple and the interpretation:

- 1: (instance of *lti*)
- 2: (num, den)
- 3: (zeros, poles, gain)
- 4: (A, B, C, D)

dt : float

The discretization time step.

method : {"gbt", "bilinear", "euler", "backward_diff", "zoh"}, optional

Which method to use:

- gbt: generalized bilinear transformation
- bilinear: Tustin's approximation ("gbt" with alpha=0.5)
- euler: Euler (or forward differencing) method ("gbt" with alpha=0)
- backward_diff: Backwards differencing ("gbt" with alpha=1.0)
- zoh: zero-order hold (default)

alpha : float within [0, 1], optional

The generalized bilinear transformation weighting parameter, which should only be specified with method="gbt", and is ignored otherwise

Returns **sysd** : tuple containing the discrete system

Based on the input type, the output will be of the form

- (num, den, dt) for transfer function input
- (zeros, poles, gain, dt) for zeros-poles-gain input

- (A, B, C, D, dt) for state-space system input

Notes

By default, the routine uses a Zero-Order Hold (zoh) method to perform the transformation. Alternatively, a generalized bilinear transformation may be used, which includes the common Tustin's bilinear approximation, an Euler's method technique, or a backwards differencing technique.

The Zero-Order Hold (zoh) method is based on *[R210]*, the generalized bilinear approximation is based on *[R211]* and *[R212]*.

References

[R210], *[R211]*, *[R212]*

`scipy.signal.place_poles`(*A*, *B*, *poles*, *method='YT'*, *rtol=0.001*, *maxiter=30*)

Compute K such that eigenvalues (A - dot(B, K))=poles.

K is the gain matrix such as the plant described by the linear system `AX+BU` will have its closed-loop poles, i.e the eigenvalues `A - B*K`, as close as possible to those asked for in poles.

SISO, MISO and MIMO systems are supported.

Parameters

A, B : ndarray

State-space representation of linear system `AX + BU`.

poles : array_like

Desired real poles and/or complex conjugates poles. Complex poles are only supported with `method="YT"` (default).

method: {'YT', 'KNV0'}, optional

Which method to choose to find the gain matrix K. One of:

- 'YT': Yang Tits
- 'KNV0': Kautsky, Nichols, Van Dooren update method 0

See References and Notes for details on the algorithms.

rtol: float, optional

After each iteration the determinant of the eigenvectors of `A - B*K` is compared to its previous value, when the relative error between these two values becomes lower than *rtol* the algorithm stops. Default is 1e-3.

maxiter: int, optional

Maximum number of iterations to compute the gain matrix. Default is 30.

Returns

full_state_feedback : Bunch object

full_state_feedback is composed of:

gain_matrix [1-D ndarray] The closed loop matrix K such as the eigenvalues of `A-BK` are as close as possible to the requested poles.

computed_poles [1-D ndarray] The poles corresponding to `A-BK` sorted as first the real poles in increasing order, then the complex congugates in lexicographic order.

requested_poles [1-D ndarray] The poles the algorithm was asked to place sorted as above, they may differ from what was achieved.

X [2-D ndarray] The transfer matrix such as `X * diag(poles) = (A - B*K)*X` (see Notes)

rtol [float] The relative tolerance achieved on `det(X)` (see Notes). *rtol* will be NaN if it is possible to solve the system `diag(poles) = (A - B*K)`, or 0 when the optimization algorithms can't do anything i.e when `B.shape[1] == 1`.

nb_iter [int] The number of iterations performed before converging. *nb_iter* will be NaN if it is possible to solve the system `diag(poles) = (A - B*K)`, or 0 when the optimization algorithms can't do anything i.e when `B.shape[1] == 1`.

Notes

The Tits and Yang (YT), *[R259]* paper is an update of the original Kautsky et al. (KNV) paper *[R258]*. KNV relies on rank-1 updates to find the transfer matrix X such that `X * diag(poles) = (A - B*K)*X`, whereas YT uses rank-2 updates. This yields on average more robust solutions (see *[R259]* pp 21-22), furthermore the YT algorithm supports complex poles whereas KNV does not in its original version. Only update method 0 proposed by KNV has been implemented here, hence the name `'KNV0'`.

KNV extended to complex poles is used in Matlab's `place` function, YT is distributed under a non-free licence by Slicot under the name `robpole`. It is unclear and undocumented how KNV0 has been extended to complex poles (Tits and Yang claim on page 14 of their paper that their method can not be used to extend KNV to complex poles), therefore only YT supports them in this implementation.

As the solution to the problem of pole placement is not unique for MIMO systems, both methods start with a tentative transfer matrix which is altered in various way to increase its determinant. Both methods have been proven to converge to a stable solution, however depending on the way the initial transfer matrix is chosen they will converge to different solutions and therefore there is absolutely no guarantee that using `'KNV0'` will yield results similar to Matlab's or any other implementation of these algorithms.

Using the default method `'YT'` should be fine in most cases; `'KNV0'` is only provided because it is needed by `'YT'` in some specific cases. Furthermore `'YT'` gives on average more robust results than `'KNV0'` when `abs(det(X))` is used as a robustness indicator.

[R259] is available as a technical report on the following URL: http://drum.lib.umd.edu/handle/1903/5598

References

[R258], *[R259]*

Examples

A simple example demonstrating real pole placement using both KNV and YT algorithms. This is example number 1 from section 4 of the reference KNV publication (*[R258]*):

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> A = np.array([[ 1.380,  -0.2077,  6.715, -5.676  ],
...               [-0.5814, -4.290,   0,      0.6750 ],
...               [ 1.067,   4.273,  -6.654,  5.893  ],
...               [ 0.0480,  4.273,   1.343, -2.104  ]])
>>> B = np.array([[ 0,      5.679 ],
...               [ 1.136,  1.136 ],
...               [ 0,      0,    ],
```

```
...                 [-3.146,   0      ]])
>>> P = np.array([-0.2, -0.5, -5.0566, -8.6659])
```

Now compute K with KNV method 0, with the default YT method and with the YT method while forcing 100 iterations of the algorithm and print some results after each call.

```
>>> fsf1 = signal.place_poles(A, B, P, method='KNV0')
>>> fsf1.gain_matrix
array([[ 0.20071427, -0.96665799,  0.24066128, -0.10279785],
       [ 0.50587268,  0.57779091,  0.51795763, -0.41991442]])
```

```
>>> fsf2 = signal.place_poles(A, B, P)  # uses YT method
>>> fsf2.computed_poles
array([-8.6659, -5.0566, -0.5   , -0.2   ])
```

```
>>> fsf3 = signal.place_poles(A, B, P, rtol=-1, maxiter=100)
>>> fsf3.X
array([[ 0.52072442+0.j, -0.08409372+0.j, -0.56847937+0.j,  0.74823657+0.j],
       [-0.04977751+0.j, -0.80872954+0.j,  0.13566234+0.j, -0.29322906+0.j],
       [-0.82266932+0.j, -0.19168026+0.j, -0.56348322+0.j, -0.43815060+0.j],
       [ 0.22267347+0.j,  0.54967577+0.j, -0.58387806+0.j, -0.40271926+0.j]])
```

The absolute value of the determinant of X is a good indicator to check the robustness of the results, both `'KNV0'` and `'YT'` aim at maximizing it. Below a comparison of the robustness of the results above:

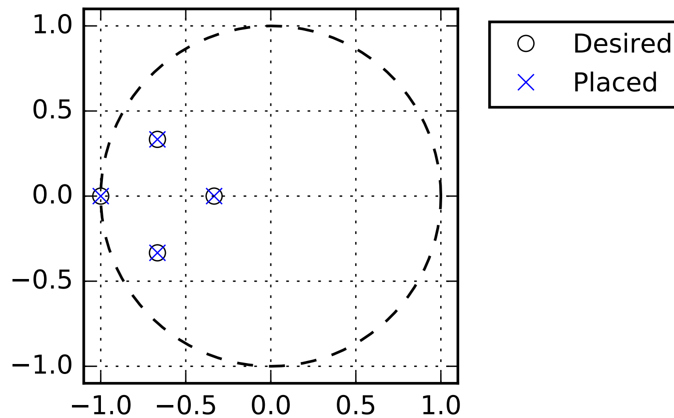
```
>>> abs(np.linalg.det(fsf1.X)) < abs(np.linalg.det(fsf2.X))
True
>>> abs(np.linalg.det(fsf2.X)) < abs(np.linalg.det(fsf3.X))
True
```

Now a simple example for complex poles:

```
>>> A = np.array([[ 0,  7/3.,  0,   0   ],
...               [ 0,   0,    0,  7/9. ],
...               [ 0,   0,    0,   0   ],
...               [ 0,   0,    0,   0   ]])
>>> B = np.array([[ 0,  0 ],
...               [ 0,  0 ],
...               [ 1,  0 ],
...               [ 0,  1 ]])
>>> P = np.array([-3, -1, -2-1j, -2+1j]) / 3.
>>> fsf = signal.place_poles(A, B, P, method='YT')
```

We can plot the desired and computed poles in the complex plane:

```
>>> t = np.linspace(0, 2*np.pi, 401)
>>> plt.plot(np.cos(t), np.sin(t), 'k--')  # unit circle
>>> plt.plot(fsf.requested_poles.real, fsf.requested_poles.imag,
...          'wo', label='Desired')
>>> plt.plot(fsf.computed_poles.real, fsf.computed_poles.imag, 'bx',
...          label='Placed')
>>> plt.grid()
>>> plt.axis('image')
>>> plt.axis([-1.1, 1.1, -1.1, 1.1])
>>> plt.legend(bbox_to_anchor=(1.05, 1), loc=2, numpoints=1)
```



5.20.9 Waveforms

`chirp`(t, f0, t1, f1[, method, phi, vertex_zero])	Frequency-swept cosine generator.
`gausspulse`(t[, fc, bw, bwr, tpr, retquad, ...])	Return a Gaussian modulated sinusoid:
`max_len_seq`(nbits[, state, length, taps])	Maximum length sequence (MLS) generator.
`sawtooth`(t[, width])	Return a periodic sawtooth or triangle waveform.
`square`(t[, duty])	Return a periodic square-wave waveform.
`sweep_poly`(t, poly[, phi])	Frequency-swept cosine generator, with a time-dependent frequency.
`unit_impulse`(shape[, idx, dtype])	Unit impulse signal (discrete delta function) or unit basis vector.

`scipy.signal.`**`chirp`**(*t*, *f0*, *t1*, *f1*, *method='linear'*, *phi=0*, *vertex_zero=True*)

Frequency-swept cosine generator.

In the following, ‘Hz’ should be interpreted as ‘cycles per unit’; there is no requirement here that the unit is one second. The important distinction is that the units of rotation are cycles, not radians. Likewise, *t* could be a measurement of space instead of time.

Parameters

t : array_like
: Times at which to evaluate the waveform.

f0 : float
: Frequency (e.g. Hz) at time t=0.

t1 : float
: Time at which *f1* is specified.

f1 : float
: Frequency (e.g. Hz) of the waveform at time *t1*.

method : {‘linear’, ‘quadratic’, ‘logarithmic’, ‘hyperbolic’}, optional
: Kind of frequency sweep. If not given, *linear* is assumed. See Notes below for more details.

phi : float, optional
: Phase offset, in degrees. Default is 0.

vertex_zero : bool, optional

This parameter is only used when *method* is 'quadratic'. It determines whether the vertex of the parabola that is the graph of the frequency is at t=0 or t=t1.

Returns **y** : ndarray

A numpy array containing the signal evaluated at *t* with the requested time-varying frequency. More precisely, the function returns `cos(phase + (pi/180)*phi)` where *phase* is the integral (from 0 to *t*) of `2*pi*f(t)`. `f(t)` is defined below.

See also:

sweep_poly

Notes

There are four options for the *method*. The following formulas give the instantaneous frequency (in Hz) of the signal generated by *chirp()*. For convenience, the shorter names shown below may also be used.

linear, lin, li:

```
f(t) = f0 + (f1 - f0) * t / t1
```

quadratic, quad, q:

The graph of the frequency f(t) is a parabola through (0, f0) and (t1, f1). By default, the vertex of the parabola is at (0, f0). If *vertex_zero* is False, then the vertex is at (t1, f1). The formula is:
if vertex_zero is True:

```
f(t) = f0 + (f1 - f0) * t**2 / t1**2
```

else:

```
f(t) = f1 - (f1 - f0) * (t1 - t)**2 / t1**2
```

To use a more general quadratic function, or an arbitrary polynomial, use the function `scipy.signal.waveforms.sweep_poly`.

logarithmic, log, lo:

```
f(t) = f0 * (f1/f0)**(t/t1)
```

f0 and f1 must be nonzero and have the same sign.
This signal is also known as a geometric or exponential chirp.

hyperbolic, hyp:

```
f(t) = f0*f1*t1 / ((f0 - f1)*t + f1*t1)
```

f0 and f1 must be nonzero.

`scipy.signal.`**`gausspulse`**(*t*, *fc=1000*, *bw=0.5*, *bwr=-6*, *tpr=-60*, *retquad=False*, *retenv=False*)

Return a Gaussian modulated sinusoid:

```
exp(-a t^2) exp(1j*2*pi*fc*t).
```

If *retquad* is True, then return the real and imaginary parts (in-phase and quadrature). If *retenv* is True, then return the envelope (unmodulated signal). Otherwise, return the real part of the modulated sinusoid.

Parameters **t** : ndarray or the string 'cutoff'

Input array.

fc : int, optional

Center frequency (e.g. Hz). Default is 1000.

bw : float, optional

Fractional bandwidth in frequency domain of pulse (e.g. Hz). Default is 0.5.

bwr : float, optional

Reference level at which fractional bandwidth is calculated (dB). Default is -6.

tpr : float, optional
: If *t* is 'cutoff', then the function returns the cutoff time for when the pulse amplitude falls below *tpr* (in dB). Default is -60.

retquad : bool, optional
: If True, return the quadrature (imaginary) as well as the real part of the signal. Default is False.

retenv : bool, optional
: If True, return the envelope of the signal. Default is False.

Returns **yI** : ndarray
: Real part of signal. Always returned.

yQ : ndarray
: Imaginary part of signal. Only returned if *retquad* is True.

yenv : ndarray
: Envelope of signal. Only returned if *retenv* is True.

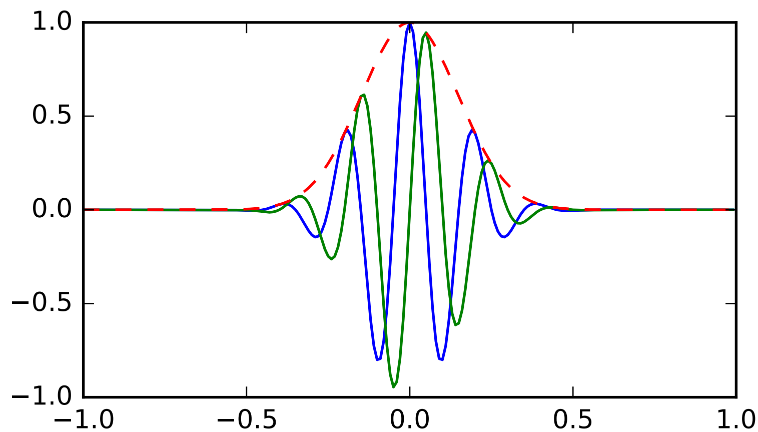
See also:

`scipy.signal.morlet`

Examples

Plot real component, imaginary component, and envelope for a 5 Hz pulse, sampled at 100 Hz for 2 seconds:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(-1, 1, 2 * 100, endpoint=False)
>>> i, q, e = signal.gausspulse(t, fc=5, retquad=True, retenv=True)
>>> plt.plot(t, i, t, q, t, e, '--')
```



`scipy.signal.`**`max_len_seq`**(*nbits*, *state=None*, *length=None*, *taps=None*)
Maximum length sequence (MLS) generator.

Parameters **nbits** : int
: Number of bits to use. Length of the resulting sequence will be `(2**nbits) - 1`. Note that generating long sequences (e.g., greater than `nbits == 16`) can take a long time.

state : array_like, optional

If array, must be of length `nbits`, and will be cast to binary (bool) representation. If None, a seed of ones will be used, producing a repeatable representation. If `state` is all zeros, an error is raised as this is invalid. Default: None.

length : int, optional

Number of samples to compute. If None, the entire length `(2**nbits) - 1` is computed.

taps : array_like, optional

Polynomial taps to use (e.g., `[7, 6, 1]` for an 8-bit sequence). If None, taps will be automatically selected (for up to `nbits == 32`).

Returns **seq** : array

Resulting MLS sequence of 0's and 1's.

state : array

The final state of the shift register.

Notes

The algorithm for MLS generation is generically described in:

https://en.wikipedia.org/wiki/Maximum_length_sequence

The default values for taps are specifically taken from the first option listed for each value of `nbits` in:

http://www.newwaveinstruments.com/resources/articles/
m_sequence_linear_feedback_shift_register_lfsr.htm

New in version 0.15.0.

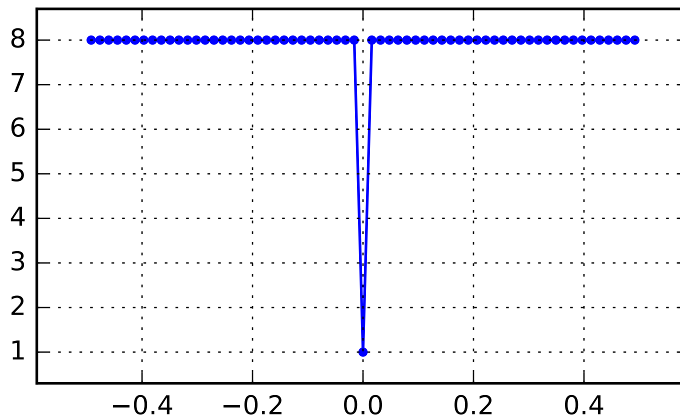
Examples

MLS uses binary convention:

```
>>> from scipy.signal import max_len_seq
>>> max_len_seq(4)[0]
array([1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0], dtype=int8)
```

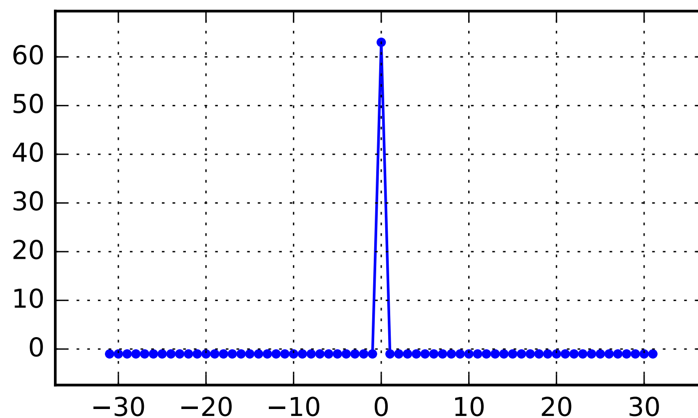
MLS has a white spectrum (except for DC):

```
>>> import matplotlib.pyplot as plt
>>> from numpy.fft import fft, ifft, fftshift, fftfreq
>>> seq = max_len_seq(6)[0]*2-1  # +1 and -1
>>> spec = fft(seq)
>>> N = len(seq)
>>> plt.plot(fftshift(fftfreq(N)), fftshift(np.abs(spec)), '.-')
>>> plt.margins(0.1, 0.1)
>>> plt.grid(True)
>>> plt.show()
```



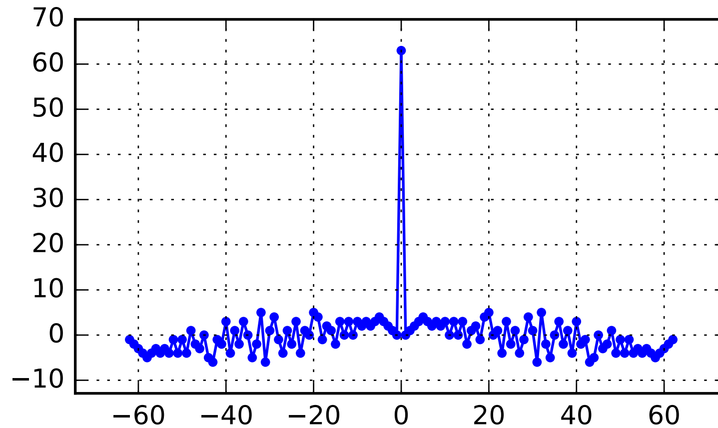
Circular autocorrelation of MLS is an impulse:

```
>>> acorrcirc = ifft(spec * np.conj(spec)).real
>>> plt.figure()
>>> plt.plot(np.arange(-N/2+1, N/2+1), fftshift(acorrcirc), '.-')
>>> plt.margins(0.1, 0.1)
>>> plt.grid(True)
>>> plt.show()
```



Linear autocorrelation of MLS is approximately an impulse:

```
>>> acorr = np.correlate(seq, seq, 'full')
>>> plt.figure()
>>> plt.plot(np.arange(-N+1, N), acorr, '.-')
>>> plt.margins(0.1, 0.1)
>>> plt.grid(True)
>>> plt.show()
```



scipy.signal.**sawtooth**(*t*, *width=1*)

Return a periodic sawtooth or triangle waveform.

The sawtooth waveform has a period `2*pi`, rises from -1 to 1 on the interval 0 to `width*2*pi`, then drops from 1 to -1 on the interval `width*2*pi` to `2*pi`. *width* must be in the interval [0, 1].

Note that this is not band-limited. It produces an infinite number of harmonics, which are aliased back and forth across the frequency spectrum.

Parameters **t** : array_like

Time.

width : array_like, optional

Width of the rising ramp as a proportion of the total cycle. Default is 1, producing a rising ramp, while 0 produces a falling ramp. *width* = 0.5 produces a triangle wave. If an array, causes wave shape to change over time, and must be the same length as t.

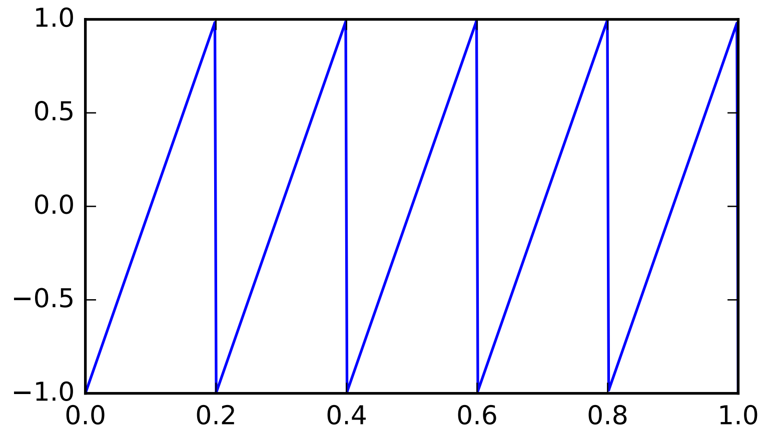
Returns **y** : ndarray

Output array containing the sawtooth waveform.

Examples

A 5 Hz waveform sampled at 500 Hz for 1 second:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(0, 1, 500)
>>> plt.plot(t, signal.sawtooth(2 * np.pi * 5 * t))
```



`scipy.signal.square(t, duty=0.5)`

Return a periodic square-wave waveform.

The square wave has a period `2*pi`, has value +1 from 0 to `2*pi*duty` and -1 from `2*pi*duty` to `2*pi`. *duty* must be in the interval [0,1].

Note that this is not band-limited. It produces an infinite number of harmonics, which are aliased back and forth across the frequency spectrum.

Parameters

t : array_like
: The input time array.

duty : array_like, optional
: Duty cycle. Default is 0.5 (50% duty cycle). If an array, causes wave shape to change over time, and must be the same length as t.

Returns

y : ndarray
: Output array containing the square waveform.

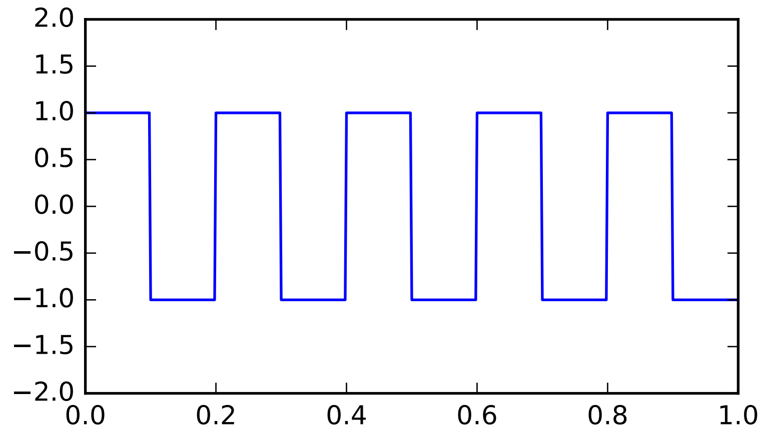
Examples

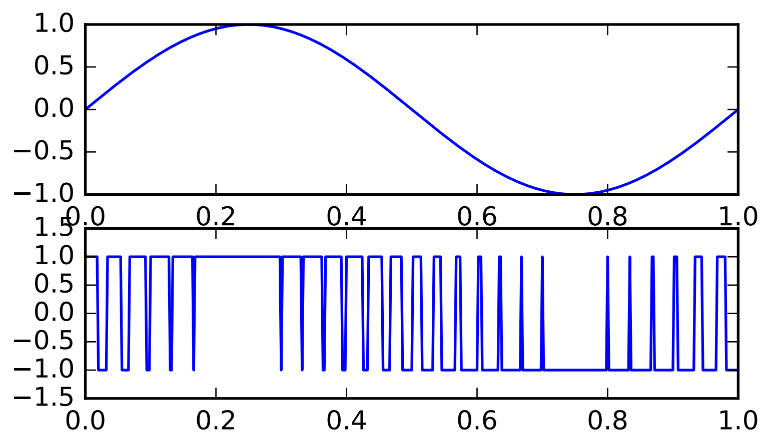
A 5 Hz waveform sampled at 500 Hz for 1 second:

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(0, 1, 500, endpoint=False)
>>> plt.plot(t, signal.square(2 * np.pi * 5 * t))
>>> plt.ylim(-2, 2)
```

A pulse-width modulated sine wave:

```
>>> plt.figure()
>>> sig = np.sin(2 * np.pi * t)
>>> pwm = signal.square(2 * np.pi * 30 * t, duty=(sig + 1)/2)
>>> plt.subplot(2, 1, 1)
>>> plt.plot(t, sig)
>>> plt.subplot(2, 1, 2)
>>> plt.plot(t, pwm)
>>> plt.ylim(-1.5, 1.5)
```





`scipy.signal.`**`sweep_poly`**(*t*, *poly*, *phi=0*)

Frequency-swept cosine generator, with a time-dependent frequency.

This function generates a sinusoidal function whose instantaneous frequency varies with time. The frequency at time *t* is given by the polynomial *poly*.

Parameters **t** : ndarray

Times at which to evaluate the waveform.

poly : 1-D array_like or instance of numpy.poly1d

The desired frequency expressed as a polynomial. If *poly* is a list or ndarray of length n, then the elements of *poly* are the coefficients of the polynomial, and the instantaneous frequency is

```
f(t) = poly[0]*t**(n-1) +
poly[1]*t**(n-2) + ... + poly[n-1]
```

If *poly* is an instance of numpy.poly1d, then the instantaneous frequency is

```
f(t) = poly(t)
```

phi : float, optional

Phase offset, in degrees, Default: 0.

Returns **sweep_poly** : ndarray

A numpy array containing the signal evaluated at *t* with the requested time-varying frequency. More precisely, the function returns `cos(phase + (pi/180)*phi)`, where *phase* is the integral (from 0 to t) of `2 * pi * f(t)`; `f(t)` is defined above.

See also:

chirp

Notes

New in version 0.8.0.

If *poly* is a list or ndarray of length *n*, then the elements of *poly* are the coefficients of the polynomial, and the instantaneous frequency is:

```
f(t) = poly[0]*t**(n-1) + poly[1]*t**(n-2) + ... + poly[n-1]
```

If *poly* is an instance of `numpy.poly1d`, then the instantaneous frequency is:

```
f(t) = poly(t)
```

Finally, the output `s` is:

```
cos(phase + (pi/180)*phi)
```

where *phase* is the integral from 0 to *t* of `2 * pi * f(t)`, `f(t)` as defined above.

`scipy.signal.`**`unit_impulse`**(*shape*, *idx=None*, *dtype=<type 'float'>*)

Unit impulse signal (discrete delta function) or unit basis vector.

Parameters **shape** : int or tuple of int

Number of samples in the output (1-D), or a tuple that represents the shape of the output (N-D).

idx : None or int or tuple of int or 'mid', optional

Index at which the value is 1. If None, defaults to the 0th element. If `idx='mid'`, the impulse will be centered at `shape // 2` in all dimensions. If an int, the impulse will be at *idx* in all dimensions.

dtype : data-type, optional

The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

Returns **y** : ndarray

Output array containing an impulse signal.

Notes

The 1D case is also known as the Kronecker delta.

New in version 0.19.0.

Examples

An impulse at the 0th element ($\delta[n]$):

```
>>> from scipy import signal
>>> signal.unit_impulse(8)
array([ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

Impulse offset by 2 samples ($\delta[n-2]$):

```
>>> signal.unit_impulse(7, 2)
array([ 0.,  0.,  1.,  0.,  0.,  0.,  0.])
```

2-dimensional impulse, centered:

```
>>> signal.unit_impulse((3, 3), 'mid')
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  0.]])
```

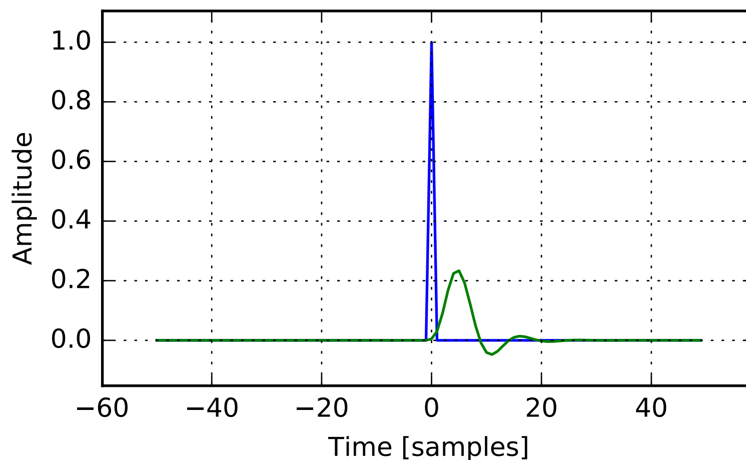
Impulse at (2, 2), using broadcasting:

```
>>> signal.unit_impulse((4, 4), 2)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Plot the impulse response of a 4th-order Butterworth lowpass filter:

```
>>> imp = signal.unit_impulse(100, 'mid')
>>> b, a = signal.butter(4, 0.2)
>>> response = signal.lfilter(b, a, imp)
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(np.arange(-50, 50), imp)
>>> plt.plot(np.arange(-50, 50), response)
>>> plt.margins(0.1, 0.1)
>>> plt.xlabel('Time [samples]')
>>> plt.ylabel('Amplitude')
>>> plt.grid(True)
>>> plt.show()
```



5.20.10 Window functions

get_window(window, Nx[, fftbins])	Return a window.
barthann(M[, sym])	Return a modified Bartlett-Hann window.
bartlett(M[, sym])	Return a Bartlett window.
blackman(M[, sym])	Return a Blackman window.
blackmanharris(M[, sym])	Return a minimum 4-term Blackman-Harris window.
bohman(M[, sym])	Return a Bohman window.
boxcar(M[, sym])	Return a boxcar or rectangular window.
chebwin(M, at[, sym])	Return a Dolph-Chebyshev window.
cosine(M[, sym])	Return a window with a simple cosine shape.
exponential(M[, center, tau, sym])	Return an exponential (or Poisson) window.
flattop(M[, sym])	Return a flat top window.
gaussian(M, std[, sym])	Return a Gaussian window.
general_gaussian(M, p, sig[, sym])	Return a window with a generalized Gaussian shape.
hamming(M[, sym])	Return a Hamming window.
hann(M[, sym])	Return a Hann window.
hanning(M[, sym])	Return a Hann window.
kaiser(M, beta[, sym])	Return a Kaiser window.
nuttall(M[, sym])	Return a minimum 4-term Blackman-Harris window according to Nuttall.
parzen(M[, sym])	Return a Parzen window.
slepian(M, width[, sym])	Return a digital Slepian (DPSS) window.
triang(M[, sym])	Return a triangular window.
tukey(M[, alpha, sym])	Return a Tukey window, also known as a tapered cosine window.

scipy.signal.**get_window**(*window*, *Nx*, *fftbins=True*)

Return a window.

Parameters

window : string, float, or tuple
: The type of window to create. See below for more details.

Nx : int
: The number of samples in the window.

fftbins : bool, optional
: If True (default), create a "periodic" window, ready to use with *ifftshift* and be multiplied by the result of an FFT (see also *fftpack.fftfreq*). If False, create a "symmetric" window, for use in filter design.

Returns

get_window : ndarray
: Returns a window of length *Nx* and type *window*

Notes

Window types:

> *boxcar*, *triang*, *blackman*, *hamming*, *hann*, *bartlett*, *flattop*, *parzen*, *bohman*, *blackmanharris*, *nuttall*, *barthann*, *kaiser* (needs beta), *gaussian* (needs standard deviation), *general_gaussian* (needs power, width), *slepian* (needs width), *chebwin* (needs attenuation), *exponential* (needs decay scale), *tukey* (needs taper fraction)

If the window requires no parameters, then *window* can be a string.

If the window requires parameters, then *window* must be a tuple with the first argument the string name of the window, and the next arguments the needed parameters.

If *window* is a floating point number, it is interpreted as the beta parameter of the *kaiser* window.

Each of the window types listed above is also the name of a function that can be called directly to create a window of that type.

Examples

```
>>> from scipy import signal
>>> signal.get_window('triang', 7)
array([ 0.125,  0.375,  0.625,  0.875,  0.875,  0.625,  0.375])
>>> signal.get_window(('kaiser', 4.0), 9)
array([ 0.08848053,  0.29425961,  0.56437221,  0.82160913,  0.97885093,
        0.97885093,  0.82160913,  0.56437221,  0.29425961])
>>> signal.get_window(4.0, 9)
array([ 0.08848053,  0.29425961,  0.56437221,  0.82160913,  0.97885093,
        0.97885093,  0.82160913,  0.56437221,  0.29425961])
```

scipy.signal.**barthann**(*M*, *sym=True*)

Return a modified Bartlett-Hann window.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

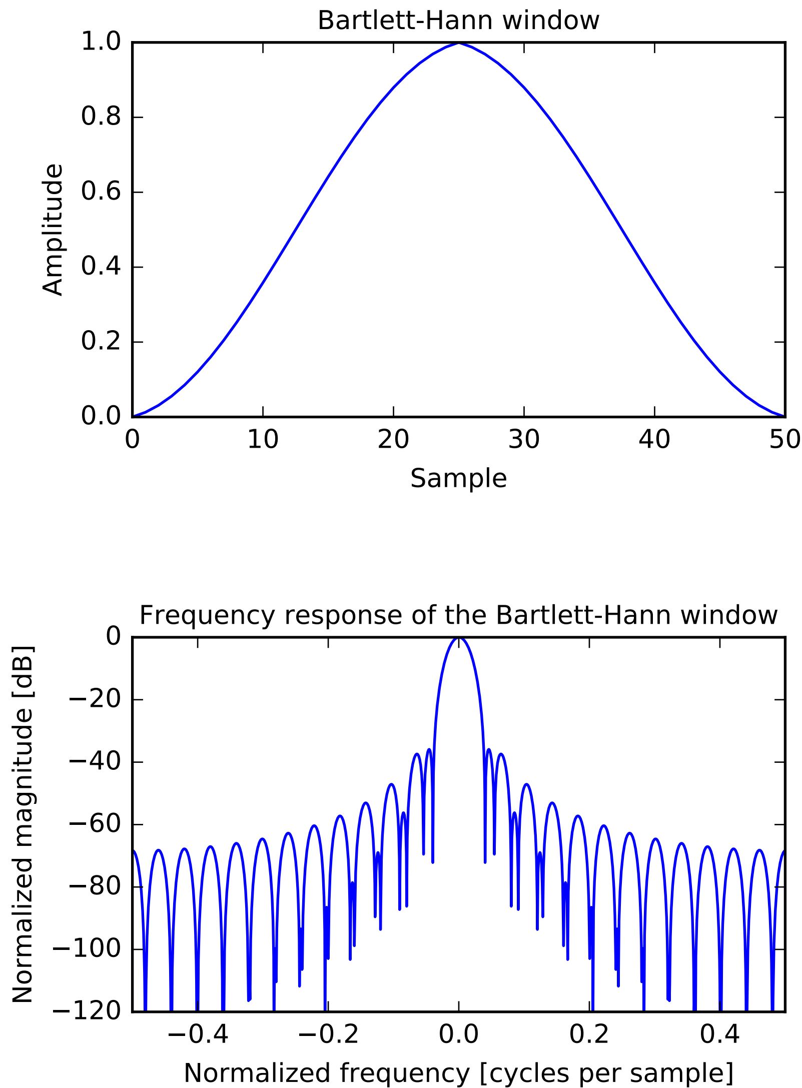
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.barthann(51)
>>> plt.plot(window)
>>> plt.title("Bartlett-Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bartlett-Hann window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.bartlett(M, sym=True)`

Return a Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The triangular window, with the first and last samples equal to zero and the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

See also:

triang A triangular window that does not touch zero at the ends

Notes

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1}\left(\frac{M-1}{2} - \left|n - \frac{M-1}{2}\right|\right)$$

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means"removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The Fourier transform of the Bartlett is the product of two sinc functions. Note the excellent discussion in Kanasewich. *[R189]*

References

[R188], *[R189]*, *[R190]*, *[R191]*, *[R192]*

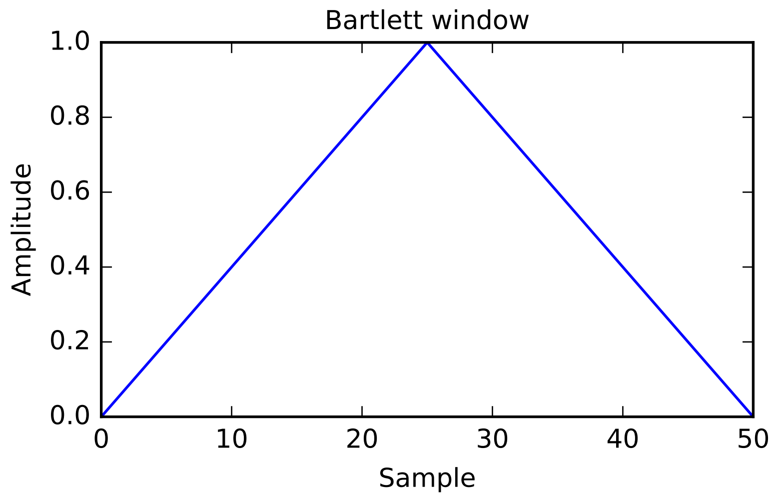
Examples

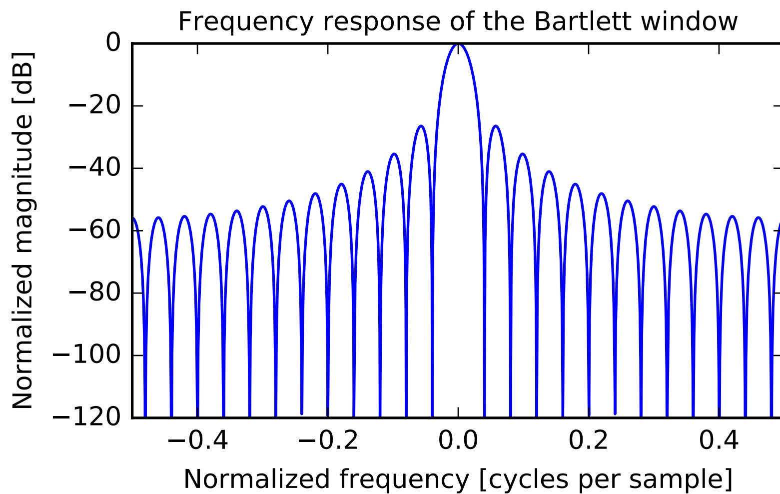
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.bartlett(51)
>>> plt.plot(window)
>>> plt.title("Bartlett window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bartlett window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.blackman`(*M*, *sym=True*)

Return a Blackman window.

The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Blackman window is defined as

$$w(n) = 0.42 - 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M)$$

The "exact Blackman" window was designed to null out the third and fourth sidelobes, but has discontinuities at the boundaries, resulting in a 6 dB/oct fall-off. This window is an approximation of the "exact" window, which does not null the sidelobes as well, but is smooth at the edges, improving the fall-off rate to 18 dB/oct. *[R202]*

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a "near optimal" tapering function, almost as good (by some measures) as the Kaiser window.

References

[R200], *[R201]*, *[R202]*

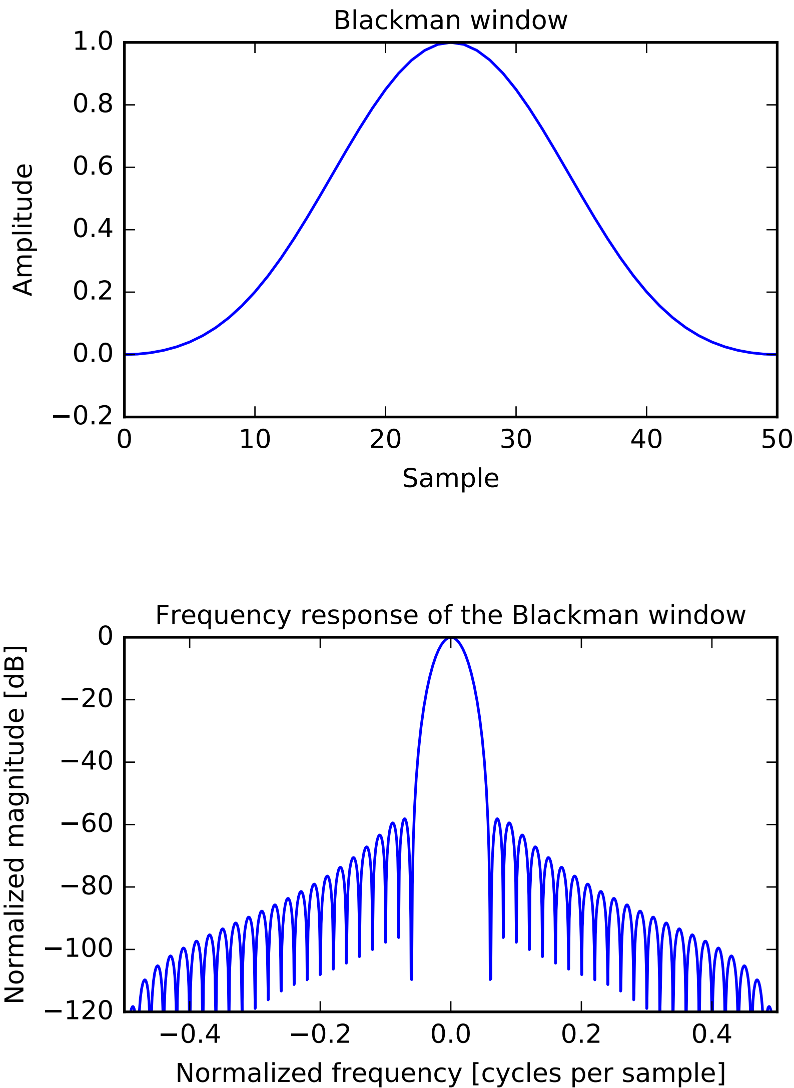
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.blackman(51)
>>> plt.plot(window)
>>> plt.title("Blackman window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Blackman window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.`**`blackmanharris`**(*M*, *sym=True*)

Return a minimum 4-term Blackman-Harris window.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

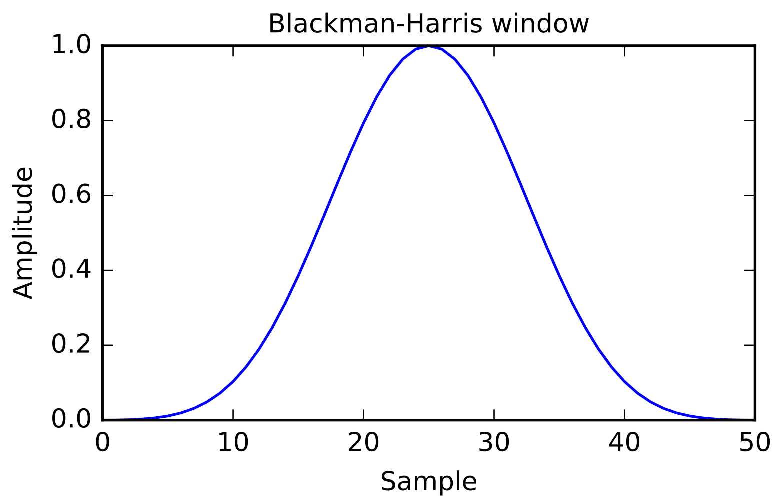
Examples

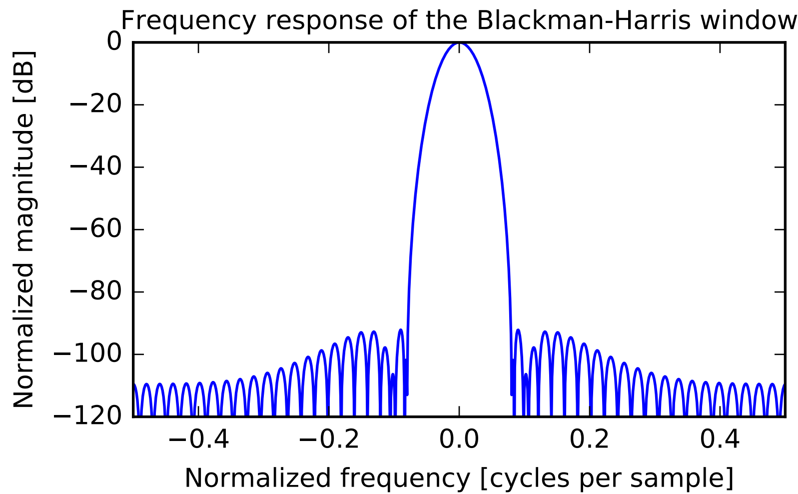
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.blackmanharris(51)
>>> plt.plot(window)
>>> plt.title("Blackman-Harris window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Blackman-Harris window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.bohman(M, sym=True)`

Return a Bohman window.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

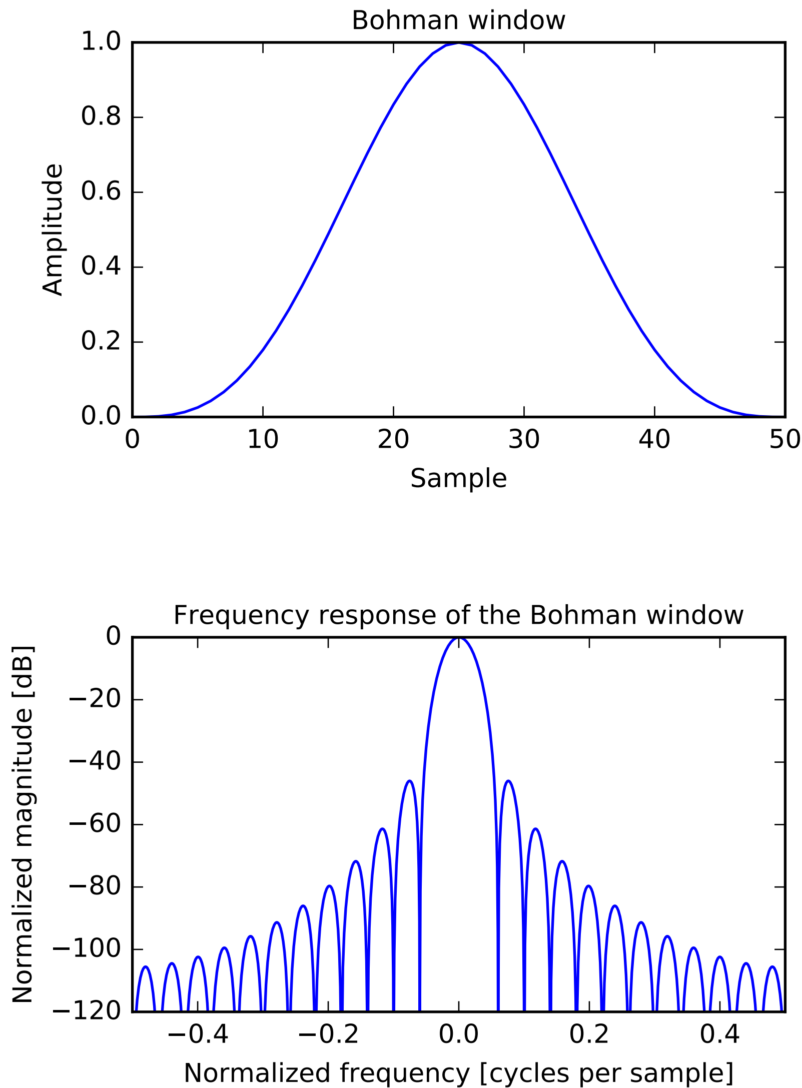
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.bohman(51)
>>> plt.plot(window)
>>> plt.title("Bohman window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Bohman window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.boxcar(M, sym=True)`

Return a boxcar or rectangular window.

Also known as a rectangular window or Dirichlet window, this is equivalent to no window at all.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

Whether the window is symmetric. (Has no effect for boxcar.)

Returns **w** : ndarray

The window, with the maximum value normalized to 1.

Examples

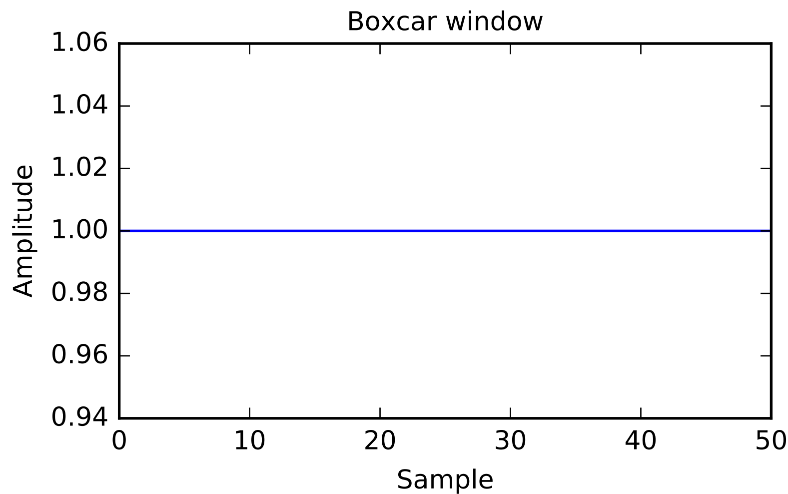
Plot the window and its frequency response:

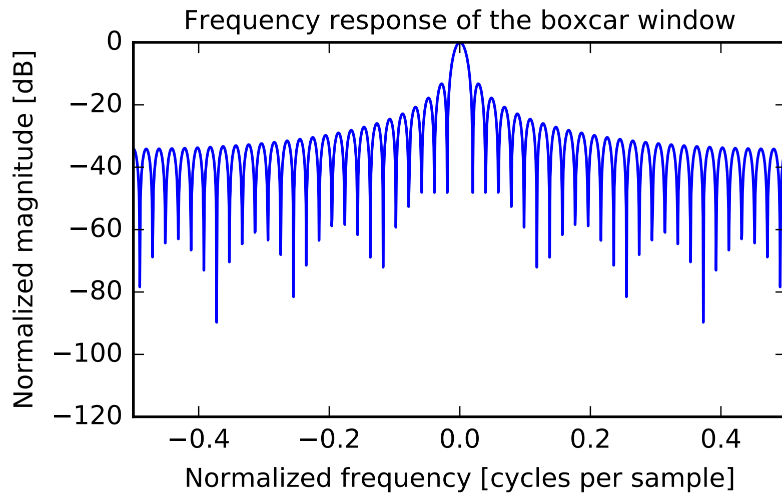
```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
```

```
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.boxcar(51)
>>> plt.plot(window)
>>> plt.title("Boxcar window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the boxcar window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.`**`chebwin`**(*M*, *at*, *sym=True*)

Return a Dolph-Chebyshev window.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

at : float

Attenuation (in dB).

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value always normalized to 1

Notes

This window optimizes for the narrowest main lobe width for a given order *M* and sidelobe equiripple attenuation *at*, using Chebyshev polynomials. It was originally developed by Dolph to optimize the directionality of radio antenna arrays.

Unlike most windows, the Dolph-Chebyshev is defined in terms of its frequency response:

$$W(k) = \frac{\cos\{M \cos^{-1}[\beta \cos(\frac{\pi k}{M})]\}}{\cosh[M \cosh^{-1}(\beta)]}$$

where

$$\beta = \cosh \left[\frac{1}{M} \cosh^{-1}(10^{\frac{A}{20}}) \right]$$

and 0 <= abs(k) <= M-1. A is the attenuation in decibels (*at*).

The time domain window is then generated using the IFFT, so power-of-two *M* are the fastest to generate, and prime number *M* are the slowest.

The equiripple condition in the frequency domain creates impulses in the time domain, which appear at the ends of the window.

References

[R203], *[R204]*, *[R205]*

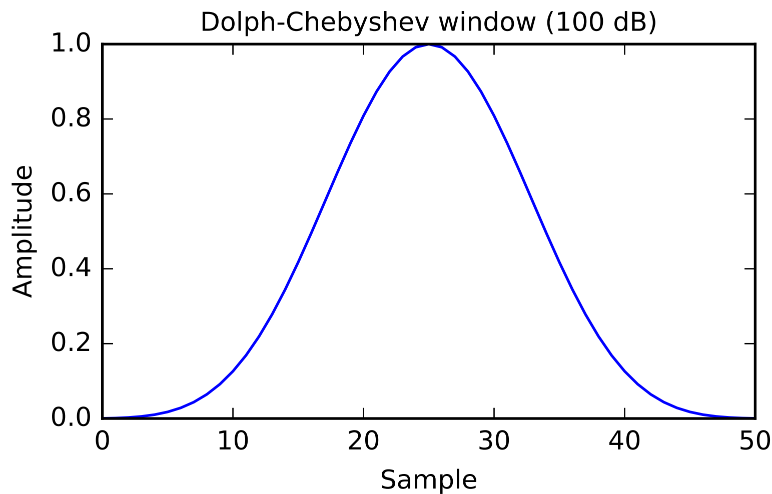
Examples

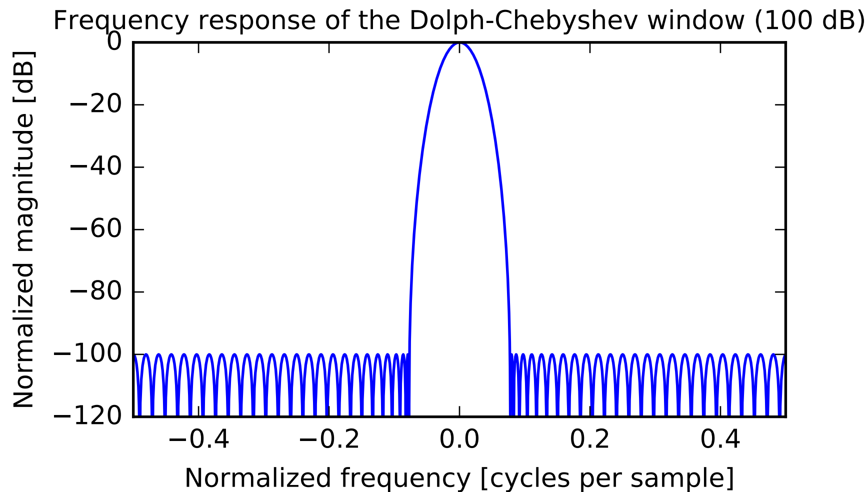
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.chebwin(51, at=100)
>>> plt.plot(window)
>>> plt.title("Dolph-Chebyshev window (100 dB)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Dolph-Chebyshev window (100 dB)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.cosine(M, sym=True)`

Return a window with a simple cosine shape.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

New in version 0.13.0.

Examples

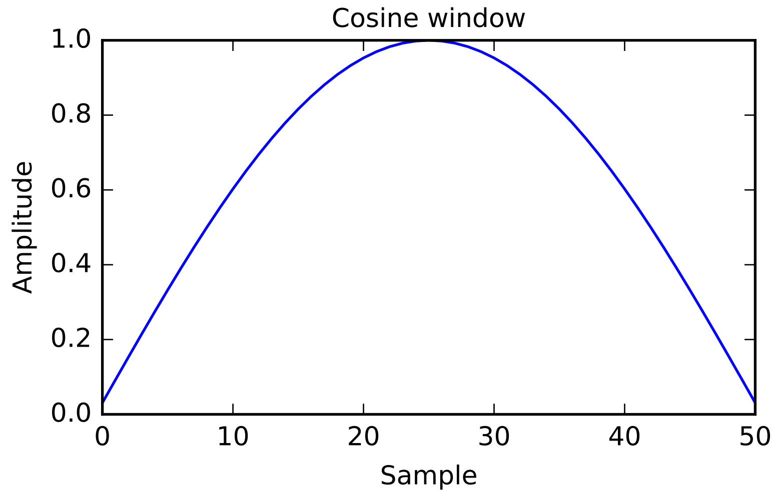
Plot the window and its frequency response:

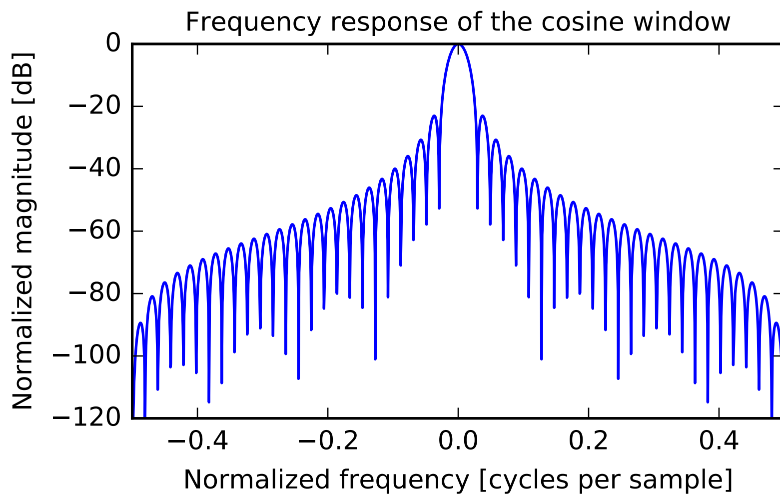
```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.cosine(51)
>>> plt.plot(window)
>>> plt.title("Cosine window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the cosine window")
```

```
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
>>> plt.show()
```





`scipy.signal.exponential`(*M*, *center=None*, *tau=1.0*, *sym=True*)

Return an exponential (or Poisson) window.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

center : float, optional

Parameter defining the center location of the window function. The default value if not given is `center = (M-1) / 2`. This parameter must take its default value for symmetric windows.

tau : float, optional

Parameter defining the decay. For `center = 0` use `tau = -(M-1) / ln(x)` if `x` is the fraction of the window remaining at the end.

sym : bool, optional
When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray
The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Exponential window is defined as

$$w(n) = e^{-|n-center|/\tau}$$

References

S. Gade and H. Herlufsen, "Windows to FFT analysis (Part I)", Technical Review 3, Bruel & Kjaer, 1987.

Examples

Plot the symmetric window and its frequency response:

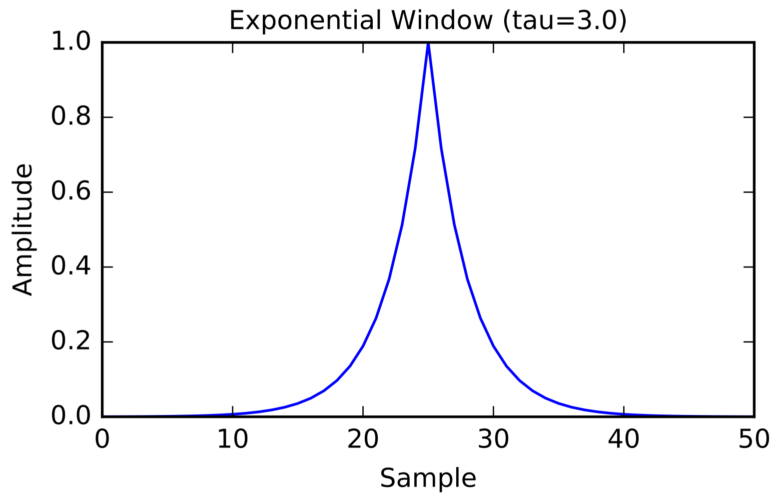
```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

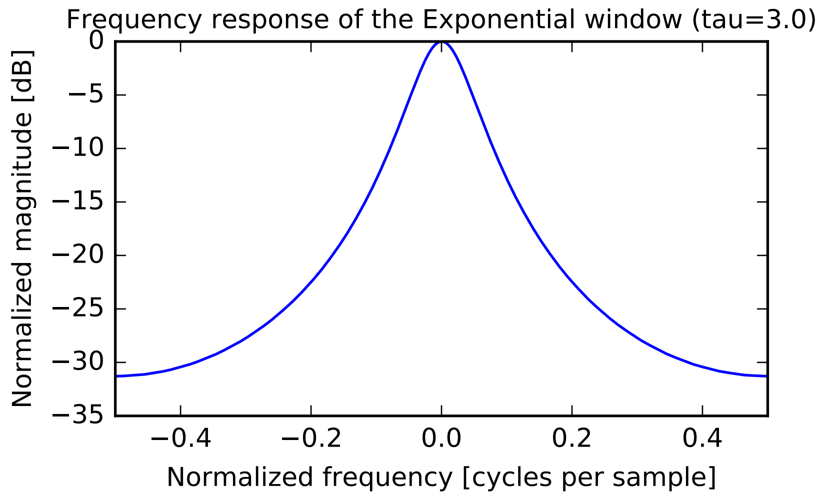
```
>>> M = 51
>>> tau = 3.0
>>> window = signal.exponential(M, tau=tau)
>>> plt.plot(window)
>>> plt.title("Exponential Window (tau=3.0)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

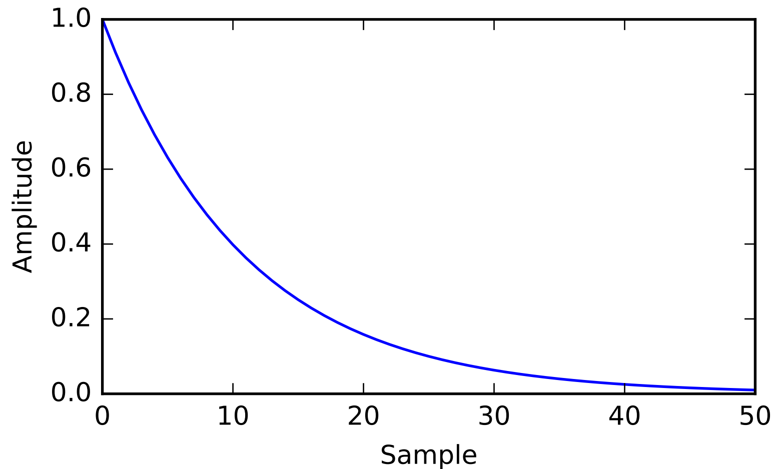
```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -35, 0])
>>> plt.title("Frequency response of the Exponential window (tau=3.0)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

This function can also generate non-symmetric windows:

```
>>> tau2 = -(M-1) / np.log(0.01)
>>> window2 = signal.exponential(M, 0, tau2, False)
>>> plt.figure()
>>> plt.plot(window2)
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```


Exponential Window (tau=3.0)
Amplitude
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
Sample


Frequency response of the Exponential window (tau=3.0)
Normalized magnitude [dB]
0
−5
−10
−15
−20
−25
−30
−35
−0.4
−0.2
0.0
0.2
0.4
Normalized frequency [cycles per sample]



`scipy.signal.flattop(M, sym=True)`

Return a flat top window.

Parameters
- **M** : int
  Number of points in the output window. If zero or less, an empty array is returned.
- **sym** : bool, optional
  When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns
- **w** : ndarray
  The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

Flat top windows are used for taking accurate measurements of signal amplitude in the frequency domain, with minimal scalloping error from the center of a frequency bin to its edges, compared to others. This is a 5th-order cosine window, with the 5 terms optimized to make the main lobe maximally flat. *[R221]*

References

[R221]

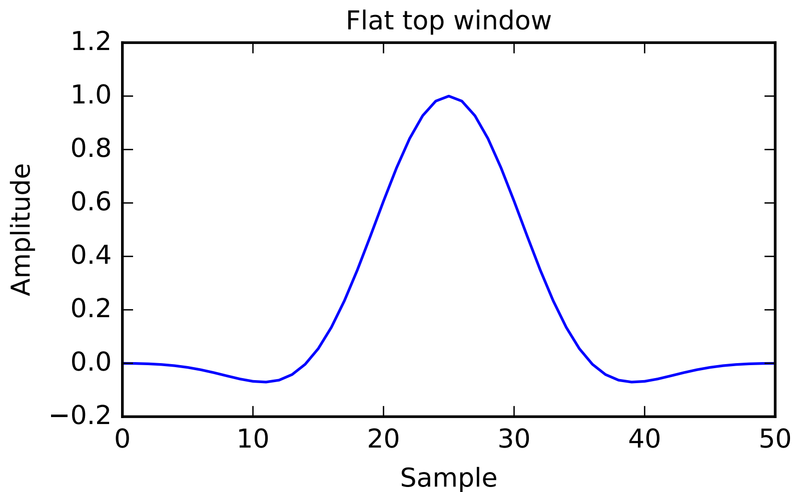
Examples

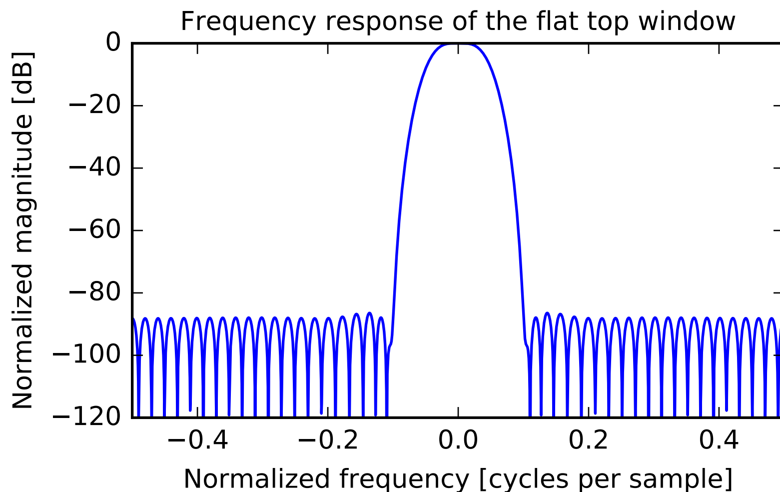
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.flattop(51)
>>> plt.plot(window)
>>> plt.title("Flat top window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the flat top window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.`**`gaussian`**(*M*, *std*, *sym=True*)

Return a Gaussian window.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

std : float

The standard deviation, sigma.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Gaussian window is defined as

$$w(n) = e^{-\frac{1}{2}\left(\frac{n}{\sigma}\right)^2}$$

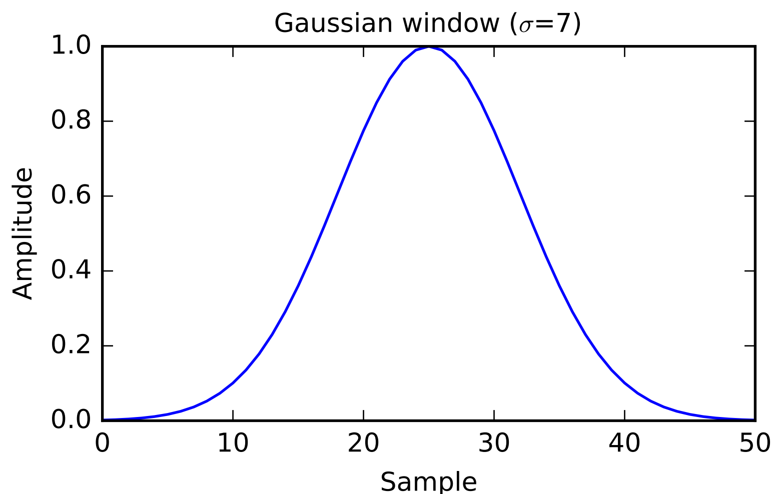
Examples

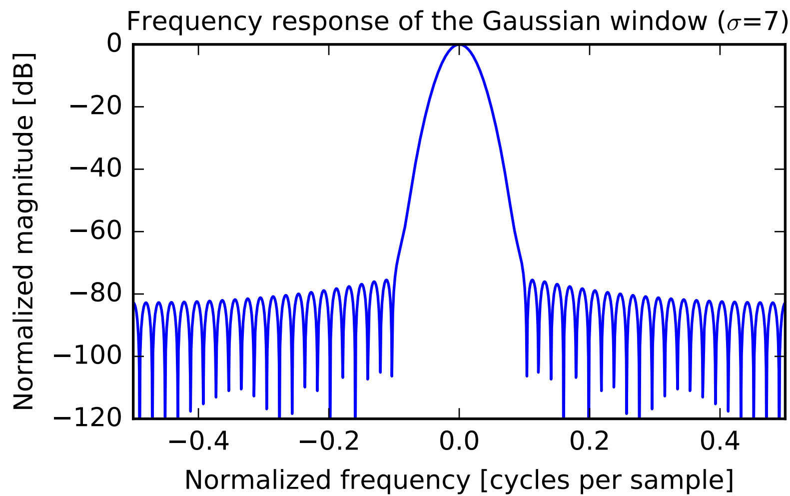
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.gaussian(51, std=7)
>>> plt.plot(window)
>>> plt.title(r"Gaussian window ($\sigma$=7)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title(r"Frequency response of the Gaussian window ($\sigma$=7)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.general_gaussian` (*M*, *p*, *sig*, *sym=True*)

Return a window with a generalized Gaussian shape.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

p : float

Shape parameter. p = 1 is identical to `gaussian`, p = 0.5 is the same shape as the Laplace distribution.

sig : float

The standard deviation, sigma.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The generalized Gaussian window is defined as

$$w(n) = e^{-\frac{1}{2}\left|\frac{n}{\sigma}\right|^{2p}}$$

the half-power point is at

$$(2\log(2))^{1/(2p)}\sigma$$

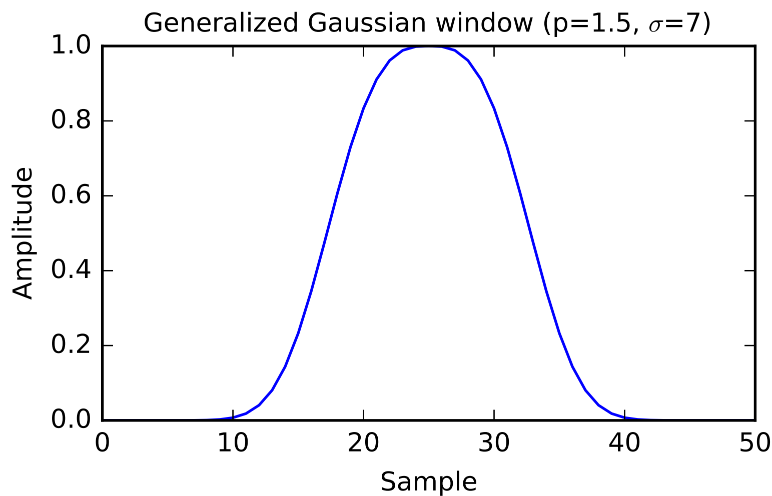
Examples

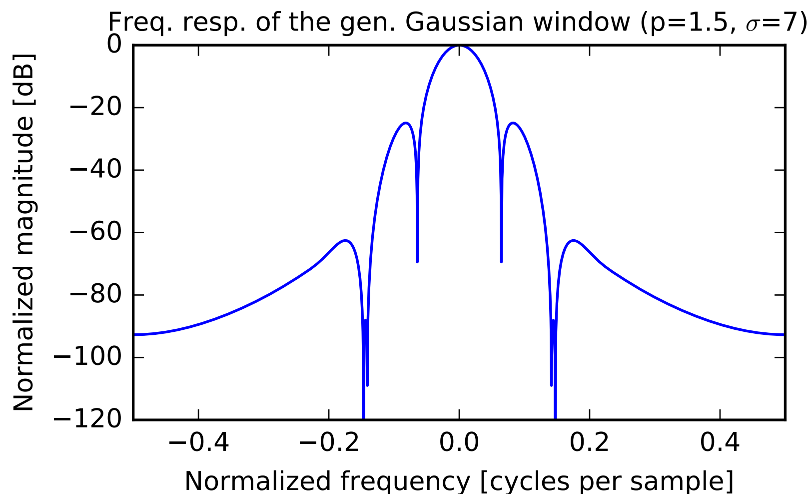
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.general_gaussian(51, p=1.5, sig=7)
>>> plt.plot(window)
>>> plt.title(r"Generalized Gaussian window (p=1.5, $\sigma$=7)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title(r"Freq. resp. of the gen. Gaussian "
...           "window (p=1.5, $\sigma$=7)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.hamming(M, sym=True)`

Return a Hamming window.

The Hamming window is a taper formed by using a raised cosine with non-zero endpoints, optimized to minimize the nearest side lobe.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Hamming window is defined as

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R223], *[R224]*, *[R225]*, *[R226]*

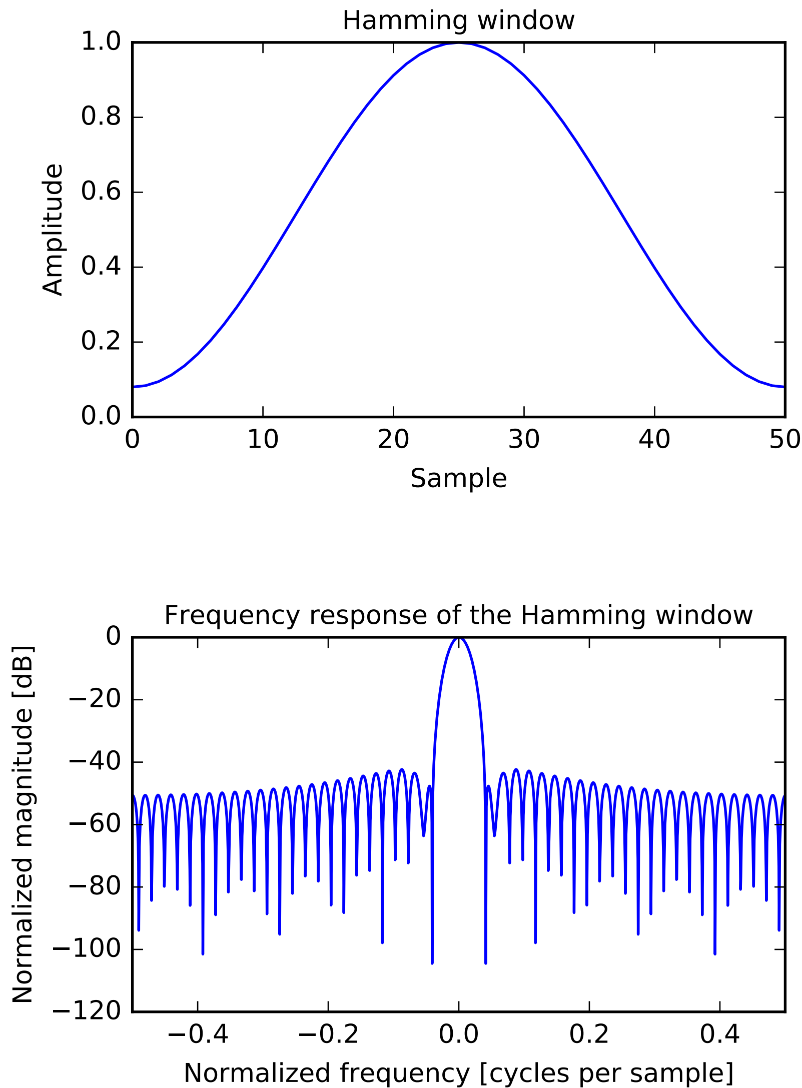
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.hamming(51)
>>> plt.plot(window)
>>> plt.title("Hamming window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Hamming window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.hann(M, sym=True)`

Return a Hann window.

The Hann window is a taper formed by using a raised cosine or sine-squared with ends that touch zero.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Hann window is defined as

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

The window was named for Julius von Hann, an Austrian meteorologist. It is also known as the Cosine Bell. It is sometimes erroneously referred to as the "Hanning" window, from the use of "hann" as a verb in the original paper and confusion with the very similar Hamming window.

Most references to the Hann window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R227], [R228], [R229], [R230]

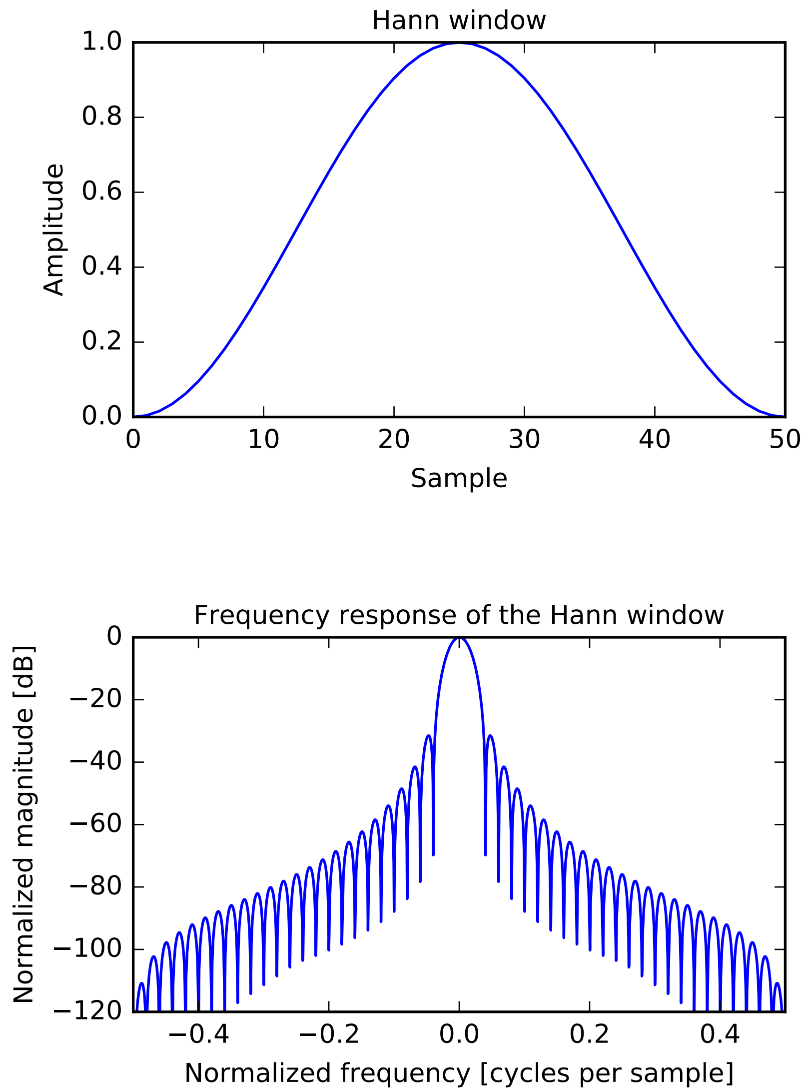
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.hann(51)
>>> plt.plot(window)
>>> plt.title("Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Hann window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.hanning(M, sym=True)`

Return a Hann window.

The Hann window is a taper formed by using a raised cosine or sine-squared with ends that touch zero.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Hann window is defined as

$$w(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

The window was named for Julius von Hann, an Austrian meteorologist. It is also known as the Cosine Bell. It is sometimes erroneously referred to as the "Hanning" window, from the use of "hann" as a verb in the original paper and confusion with the very similar Hamming window.

Most references to the Hann window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R231], [R232], [R233], [R234]

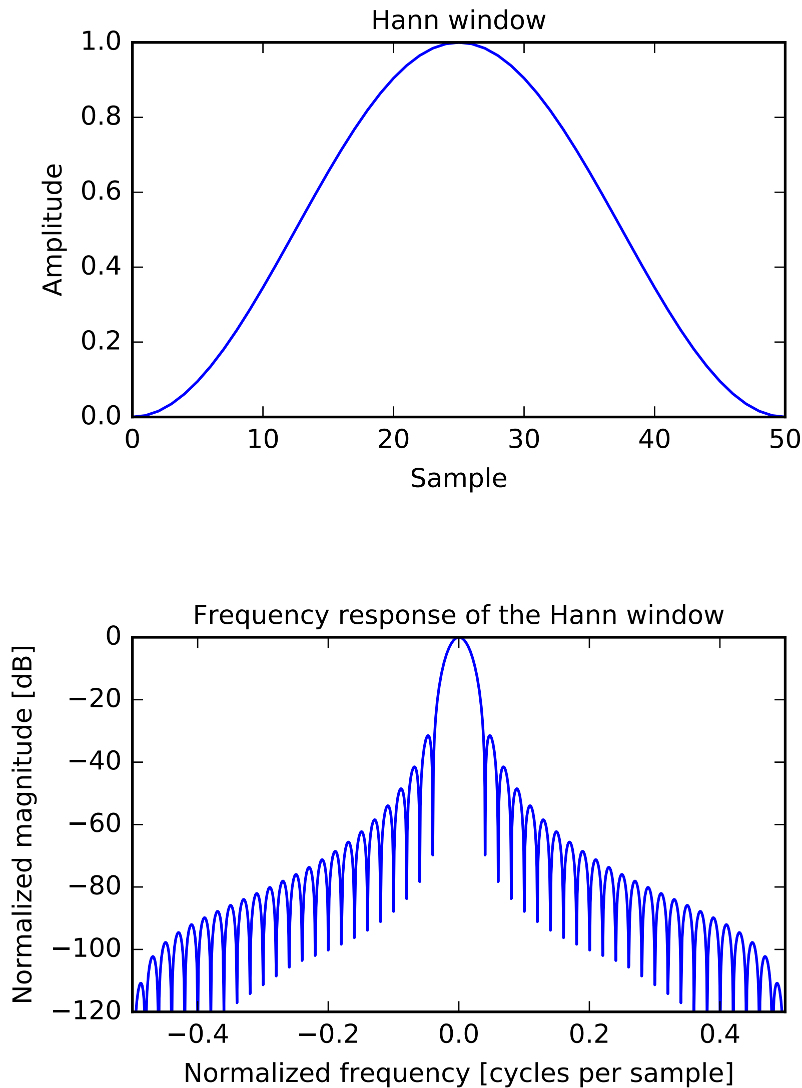
Examples

Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.hann(51)
>>> plt.plot(window)
>>> plt.title("Hann window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Hann window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.kaiser(M, beta, sym=True)`

Return a Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

beta : float

Shape parameter, determines trade-off between main-lobe width and side lobe level. As beta gets large, the window narrows.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

Notes

The Kaiser window is defined as

$$w(n) = I_0\left(\beta\sqrt{1-\frac{4n^2}{(M-1)^2}}\right)/I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2},$$

where I_0 is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple approximation to the DPSS window based on Bessel functions. The Kaiser window is a very good approximation to the Digital Prolate Spheroidal Sequence, or Slepian window, which is the transform which maximizes the energy in the main lobe of the window relative to total energy.

The Kaiser can approximate other windows by varying the beta parameter. (Some literature uses alpha = beta/pi.) *[R246]*

beta	Window shape
0	Rectangular
5	Similar to a Hamming
6	Similar to a Hann
8.6	Similar to a Blackman

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise NaNs will be returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

References

[R243], *[R244]*, *[R245]*, *[R246]*

Examples

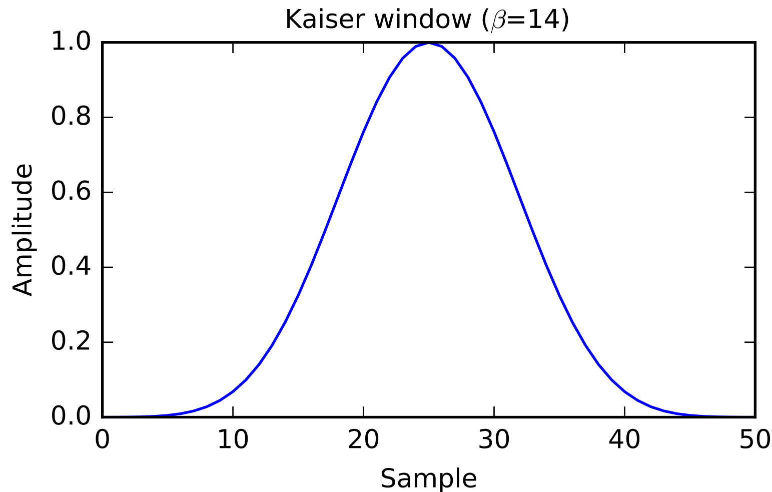
Plot the window and its frequency response:

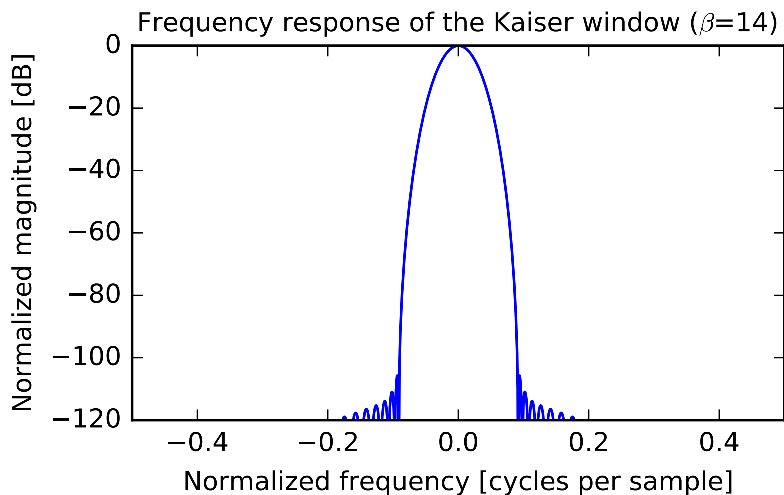
```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.kaiser(51, beta=14)
>>> plt.plot(window)
>>> plt.title(r"Kaiser window ($\beta$=14)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
```

```
>>> plt.title(r"Frequency response of the Kaiser window ($\beta$=14)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.nuttall(M, sym=True)`

Return a minimum 4-term Blackman-Harris window according to Nuttall.

This variation is called "Nuttall4c" by Heinzel. *[R256]*

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

References

[R255], [R256]

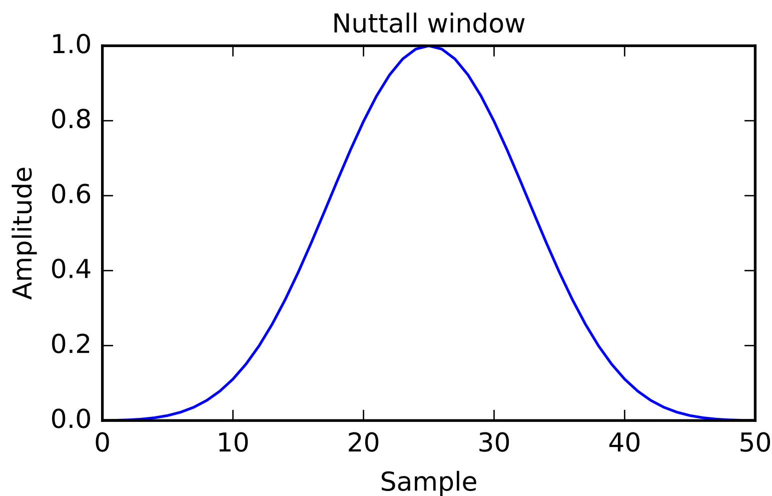
Examples

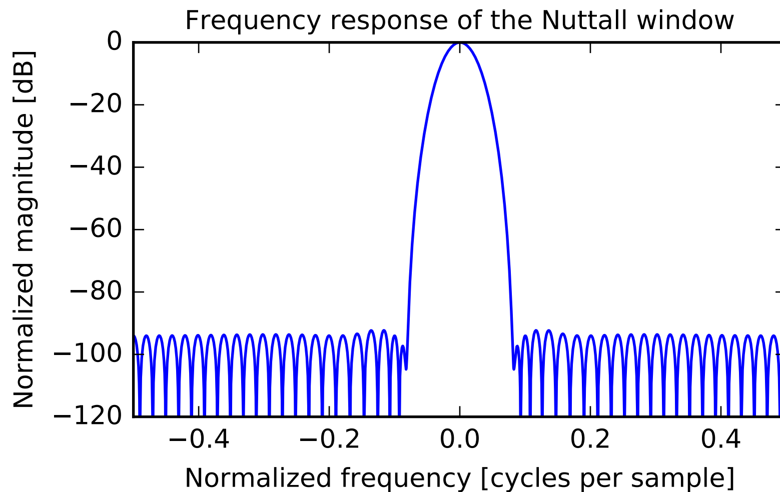
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.nuttall(51)
>>> plt.plot(window)
>>> plt.title("Nuttall window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Nuttall window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.parzen`(*M*, *sym=True*)

Return a Parzen window.

Parameters

M : int

Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

References

[R257]

Examples

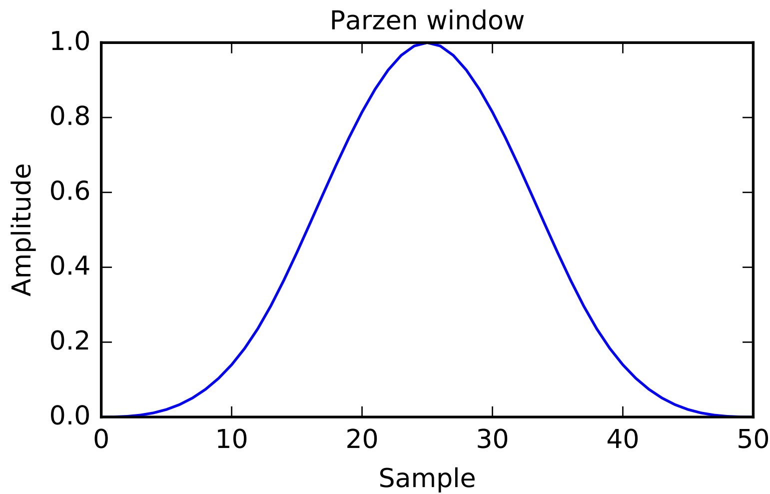
Plot the window and its frequency response:

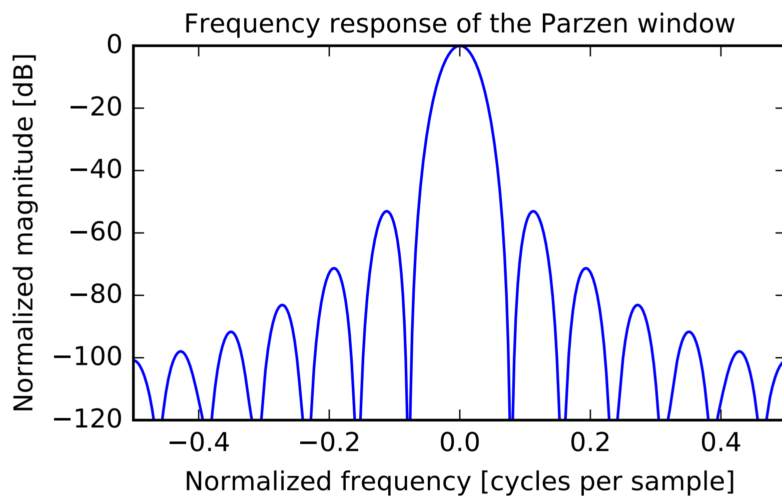
```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.parzen(51)
>>> plt.plot(window)
>>> plt.title("Parzen window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Parzen window")
```

```
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.slepian(M, width, sym=True)`

Return a digital Slepian (DPSS) window.

Used to maximize the energy concentration in the main lobe. Also called the digital prolate spheroidal sequence (DPSS).

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

width : float

Bandwidth

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value always normalized to 1

References

[R262], [R263]

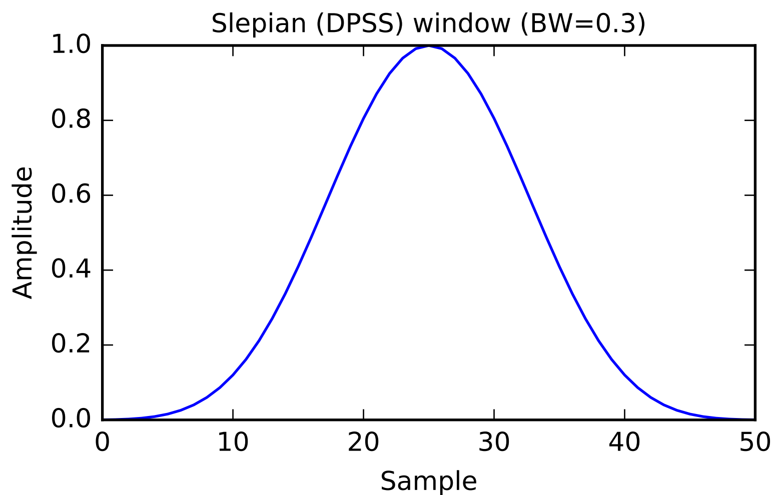
Examples

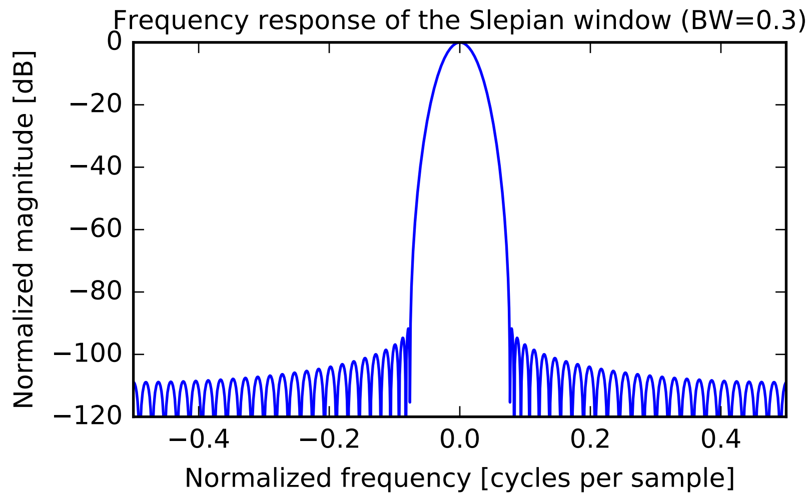
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.slepian(51, width=0.3)
>>> plt.plot(window)
>>> plt.title("Slepian (DPSS) window (BW=0.3)")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Slepian window (BW=0.3)")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





`scipy.signal.triang(M, sym=True)`

Return a triangular window.

Parameters

M : int
: Number of points in the output window. If zero or less, an empty array is returned.

sym : bool, optional
: When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns

w : ndarray
: The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

See also:

bartlett A triangular window that touches zero

Examples

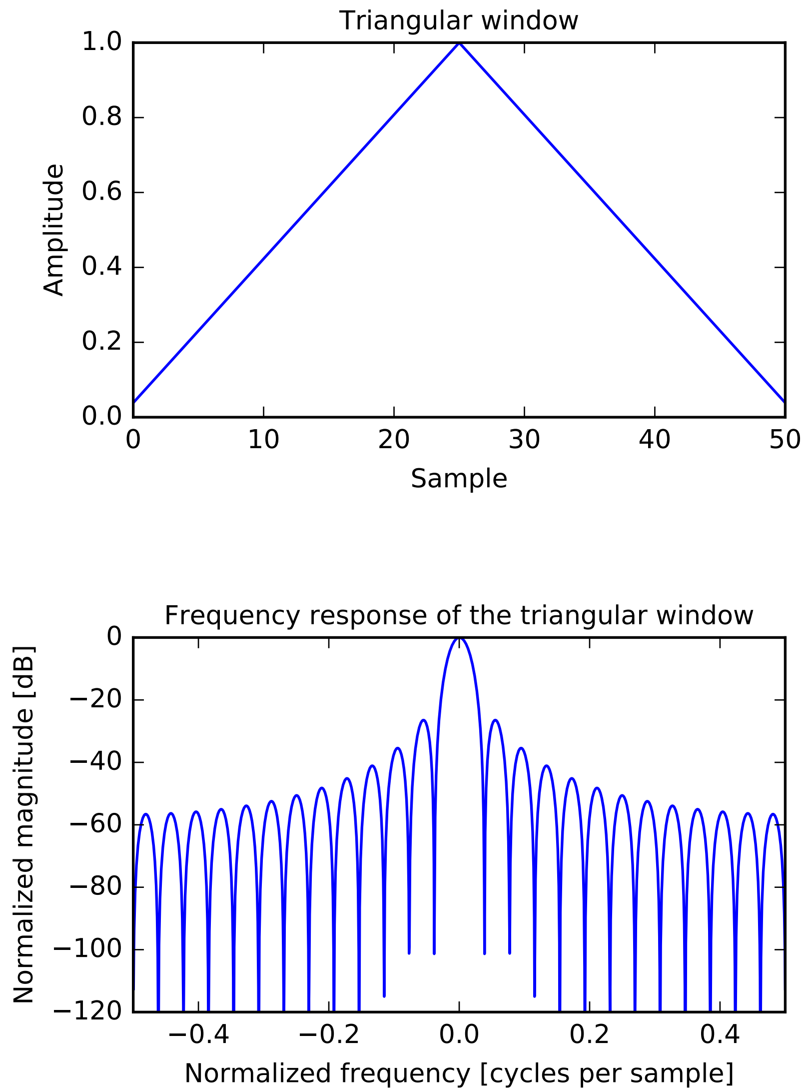
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.triang(51)
>>> plt.plot(window)
>>> plt.title("Triangular window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the triangular window")
```

```
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```



`scipy.signal.tukey`(*M*, *alpha=0.5*, *sym=True*)

Return a Tukey window, also known as a tapered cosine window.

Parameters **M** : int

Number of points in the output window. If zero or less, an empty array is returned.

alpha : float, optional

Shape parameter of the Tukey window, representing the fraction of the window inside the cosine tapered region. If zero, the Tukey window is equivalent to a rectangular window. If one, the Tukey window is equivalent to a Hann window.

sym : bool, optional

When True (default), generates a symmetric window, for use in filter design. When False, generates a periodic window, for use in spectral analysis.

Returns **w** : ndarray

The window, with the maximum value normalized to 1 (though the value 1 does not appear if *M* is even and *sym* is True).

References

[R267], [R268]

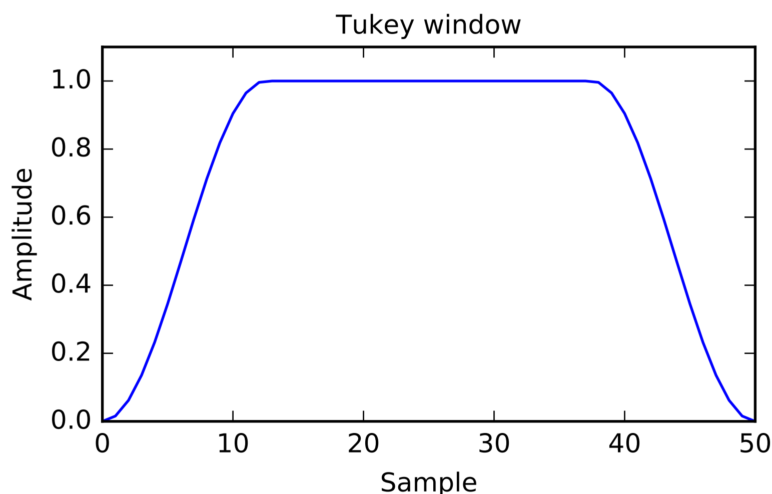
Examples

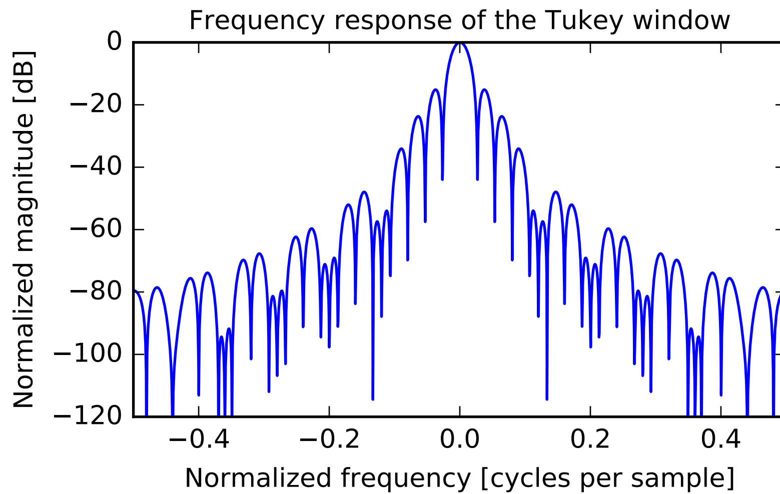
Plot the window and its frequency response:

```
>>> from scipy import signal
>>> from scipy.fftpack import fft, fftshift
>>> import matplotlib.pyplot as plt
```

```
>>> window = signal.tukey(51)
>>> plt.plot(window)
>>> plt.title("Tukey window")
>>> plt.ylabel("Amplitude")
>>> plt.xlabel("Sample")
>>> plt.ylim([0, 1.1])
```

```
>>> plt.figure()
>>> A = fft(window, 2048) / (len(window)/2.0)
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
>>> plt.plot(freq, response)
>>> plt.axis([-0.5, 0.5, -120, 0])
>>> plt.title("Frequency response of the Tukey window")
>>> plt.ylabel("Normalized magnitude [dB]")
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```





5.20.11 Wavelets

cascade(hk[, J])	Return (x, phi, psi) at dyadic points `K/2**J` from filter coefficients.
daub(p)	The coefficients for the FIR low-pass filter producing Daubechies wavelets.
morlet(M[, w, s, complete])	Complex Morlet wavelet.
qmf(hk)	Return high-pass qmf filter from low-pass
ricker(points, a)	Return a Ricker wavelet, also known as the "Mexican hat wavelet".
cwt(data, wavelet, widths)	Continuous wavelet transform.

`scipy.signal.`**`cascade`**(*hk*, *J=7*)

Return (x, phi, psi) at dyadic points `K/2**J` from filter coefficients.

Parameters

hk : array_like
: Coefficients of low-pass filter.

J : int, optional
: Values will be computed at grid points `K/2**J`. Default is 7.

Returns

x : ndarray
: The dyadic points `K/2**J` for `K=0...N * (2**J)-1` where `len(hk) = len(gk) = N+1`.

phi : ndarray
: The scaling function `phi(x)` at *x*: `phi(x) = sum(hk * phi(2x-k))`, where k is from 0 to N.

psi : ndarray, optional
: The wavelet function `psi(x)` at *x*: `phi(x) = sum(gk * phi(2x-k))`, where k is from 0 to N. *psi* is only returned if *gk* is not None.

Notes

The algorithm uses the vector cascade algorithm described by Strang and Nguyen in "Wavelets and Filter Banks". It builds a dictionary of values and slices for quick reuse. Then inserts vectors into final vector at

the end.

`scipy.signal.daub(p)`

The coefficients for the FIR low-pass filter producing Daubechies wavelets.

p>=1 gives the order of the zero at f=1/2. There are 2p filter coefficients.

Parameters **p** : int

Order of the zero at f=1/2, can have values from 1 to 34.

Returns **daub** : ndarray

Return

`scipy.signal.morlet(M, w=5.0, s=1.0, complete=True)`

Complex Morlet wavelet.

Parameters **M** : int

Length of the wavelet.

w : float, optional

Omega0. Default is 5

s : float, optional

Scaling factor, windowed from `-s*2*pi` to `+s*2*pi`. Default is 1.

complete : bool, optional

Whether to use the complete or the standard version.

Returns **morlet** : (M,) ndarray

See also:

scipy.signal.gausspulse

Notes

The standard version:

```
pi**-0.25 * exp(1j*w*x) * exp(-0.5*(x**2))
```

This commonly used wavelet is often referred to simply as the Morlet wavelet. Note that this simplified version can cause admissibility problems at low values of *w*.

The complete version:

```
pi**-0.25 * (exp(1j*w*x) - exp(-0.5*(w**2))) * exp(-0.5*(x**2))
```

This version has a correction term to improve admissibility. For *w* greater than 5, the correction term is negligible.

Note that the energy of the return wavelet is not normalised according to `s`.

The fundamental frequency of this wavelet in Hz is given by `f = 2*s*w*r / M` where *r* is the sampling rate.

Note: This function was created before *cwt* and is not compatible with it.

`scipy.signal.qmf(hk)`

Return high-pass qmf filter from low-pass

Parameters **hk** : array_like

Coefficients of high-pass filter.

`scipy.signal.ricker(points, a)`

Return a Ricker wavelet, also known as the “Mexican hat wavelet”.

It models the function:

```
A (1 - x^2/a^2) exp(-x^2/2 a^2),
```

where `A = 2/sqrt(3a)pi^1/4`.

Parameters **points** : int

Number of points in *vector*. Will be centered around 0.

a : scalar

Width parameter of the wavelet.

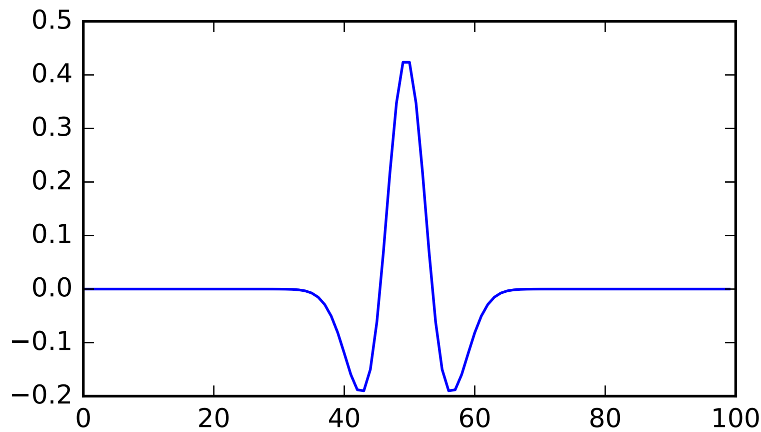
Returns **vector** : (N,) ndarray

Array of length *points* in shape of ricker curve.

Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

```
>>> points = 100
>>> a = 4.0
>>> vec2 = signal.ricker(points, a)
>>> print(len(vec2))
100
>>> plt.plot(vec2)
>>> plt.show()
```



`scipy.signal.cwt(data, wavelet, widths)`

Continuous wavelet transform.

Performs a continuous wavelet transform on *data*, using the *wavelet* function. A CWT performs a convolution with *data* using the *wavelet* function, which is characterized by a width parameter and length parameter.

Parameters **data** : (N,) ndarray

data on which to perform the transform.

wavelet : function

Wavelet function, which should take 2 arguments. The first argument is the number of points that the returned vector will have (len(wavelet(length,width)) == length). The second is a width parameter, defining the size of the wavelet (e.g. standard deviation of a gaussian). See `ricker`, which satisfies these requirements.

widths : (M,) sequence

Widths to use for transform.

Returns cwt: (M, N) ndarray

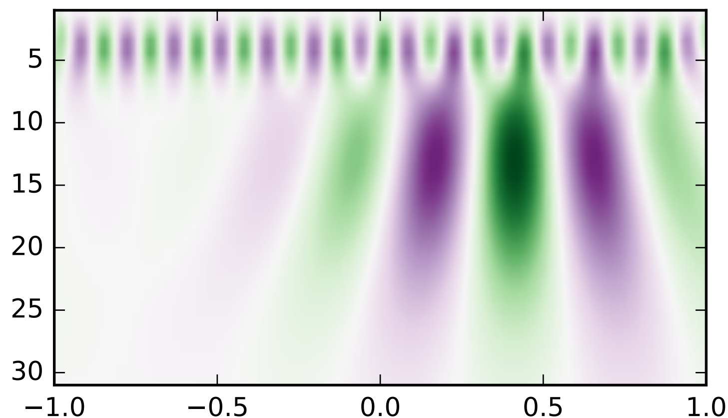
Will have shape of (len(widths), len(data)).

Notes

```
length = min(10 * width[ii], len(data))
cwt[ii,:] = signal.convolve(data, wavelet(length,
                            width[ii]), mode='same')
```

Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(-1, 1, 200, endpoint=False)
>>> sig  = np.cos(2 * np.pi * 7 * t) + signal.gausspulse(t - 0.4, fc=2)
>>> widths = np.arange(1, 31)
>>> cwtmatr = signal.cwt(sig, signal.ricker, widths)
>>> plt.imshow(cwtmatr, extent=[-1, 1, 31, 1], cmap='PRGn', aspect='auto',
...            vmax=abs(cwtmatr).max(), vmin=-abs(cwtmatr).max())
>>> plt.show()
```



5.20.12 Peak finding

find_peaks_cwt(vector, widths[, wavelet, ...])	Attempt to find the peaks in a 1-D array.
argrelmin(data[, axis, order, mode])	Calculate the relative minima of *data*.
argrelmax(data[, axis, order, mode])	Calculate the relative maxima of *data*.
argrelextrema(data, comparator[, axis, ...])	Calculate the relative extrema of *data*.

scipy.signal.**find_peaks_cwt**(*vector*, *widths*, *wavelet=None*, *max_distances=None*, *gap_thresh=None*, *min_length=None*, *min_snr=1*, *noise_perc=10*)

Attempt to find the peaks in a 1-D array.

The general approach is to smooth *vector* by convolving it with *wavelet(width)* for each width in *widths*. Relative maxima which appear at enough length scales, and with sufficiently high SNR, are accepted.

Parameters **vector** : ndarray

1-D array in which to find the peaks.

widths : sequence
: 1-D array of widths to use for calculating the CWT matrix. In general, this range should cover the expected width of peaks of interest.

wavelet : callable, optional
: Should take two parameters and return a 1-D array to convolve with *vector*. The first parameter determines the number of points of the returned wavelet array, the second parameter is the scale (*width*) of the wavelet. Should be normalized and symmetric. Default is the ricker wavelet.

max_distances : ndarray, optional
: At each row, a ridge line is only connected if the relative max at row[n] is within `max_distances[n]` from the relative max at `row[n+1]`. Default value is `widths/4`.

gap_thresh : float, optional
: If a relative maximum is not found within *max_distances*, there will be a gap. A ridge line is discontinued if there are more than *gap_thresh* points without connecting a new relative maximum. Default is 2.

min_length : int, optional
: Minimum length a ridge line needs to be acceptable. Default is `cwt.shape[0] / 4`, ie 1/4-th the number of widths.

min_snr : float, optional
: Minimum SNR ratio. Default 1. The signal is the value of the cwt matrix at the shortest length scale (`cwt[0, loc]`), the noise is the *noise_perc'th percentile of datapoints contained within a window of 'window_size* around `cwt[0, loc]`.

noise_perc : float, optional
: When calculating the noise floor, percentile of data points examined below which to consider noise. Calculated using *stats.scoreatpercentile*. Default is 10.

Returns

peaks_indices : ndarray
: Indices of the locations in the *vector* where peaks were found. The list is sorted.

See also:

cwt

Notes

This approach was designed for finding sharp peaks among noisy data, however with proper parameter selection it should function well for different peak shapes.

The algorithm is as follows:

1. Perform a continuous wavelet transform on *vector*, for the supplied *widths*. This is a convolution of *vector* with *wavelet(width)* for each width in *widths*. See *cwt*
2. Identify "ridge lines" in the cwt matrix. These are relative maxima at each row, connected across adjacent rows. See identify_ridge_lines
3. Filter the ridge_lines using filter_ridge_lines.

New in version 0.11.0.

References

[R217]

Examples

```
>>> from scipy import signal
>>> xs = np.arange(0, np.pi, 0.05)
>>> data = np.sin(xs)
>>> peakind = signal.find_peaks_cwt(data, np.arange(1,10))
>>> peakind, xs[peakind], data[peakind]
([32], array([ 1.6]), array([ 0.9995736]))
```

`scipy.signal.argrelmin`(*data*, *axis=0*, *order=1*, *mode='clip'*)

Calculate the relative minima of *data*.

Parameters

data : ndarray

Array in which to find the relative minima.

axis : int, optional

Axis over which to select from *data*. Default is 0.

order : int, optional

How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.

mode : str, optional

How the edges of the vector are treated. Available options are 'wrap' (wrap around) or 'clip' (treat overflow as the same as the last (or first) element). Default 'clip'. See numpy.take

Returns

extrema : tuple of ndarrays

Indices of the minima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

See also:

argrelextrema, *argrelmax*

Notes

This function uses *argrelextrema* with np.less as comparator.

New in version 0.11.0.

Examples

```
>>> from scipy.signal import argrelmin
>>> x = np.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelmin(x)
(array([1, 5]),)
>>> y = np.array([[1, 2, 1, 2],
...               [2, 2, 0, 0],
...               [5, 3, 4, 4]])
...
>>> argrelmin(y, axis=1)
(array([0, 2]), array([2, 1]))
```

`scipy.signal.argrelmax`(*data*, *axis=0*, *order=1*, *mode='clip'*)

Calculate the relative maxima of *data*.

Parameters

data : ndarray

Array in which to find the relative maxima.

axis : int, optional

Axis over which to select from *data*. Default is 0.

order : int, optional

How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.

mode : str, optional
How the edges of the vector are treated. Available options are 'wrap' (wrap around) or 'clip' (treat overflow as the same as the last (or first) element). Default 'clip'. See `numpy.take`.

Returns **extrema** : tuple of ndarrays
Indices of the maxima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

See also:

argrelextrema, *argrelmin*

Notes

This function uses *argrelextrema* with np.greater as comparator.

New in version 0.11.0.

Examples

```
>>> from scipy.signal import argrelmax
>>> x = np.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelmax(x)
(array([3, 6]),)
>>> y = np.array([[1, 2, 1, 2],
...               [2, 2, 0, 0],
...               [5, 3, 4, 4]])
...
>>> argrelmax(y, axis=1)
(array([0]), array([1]))
```

`scipy.signal.`**`argrelextrema`**(*data*, *comparator*, *axis=0*, *order=1*, *mode='clip'*)
Calculate the relative extrema of *data*.

Parameters **data** : ndarray
Array in which to find the relative extrema.

comparator : callable
Function to use to compare two data points. Should take two arrays as arguments.

axis : int, optional
Axis over which to select from *data*. Default is 0.

order : int, optional
How many points on each side to use for the comparison to consider `comparator(n, n+x)` to be True.

mode : str, optional
How the edges of the vector are treated. 'wrap' (wrap around) or 'clip' (treat overflow as the same as the last (or first) element). Default is 'clip'. See `numpy.take`.

Returns **extrema** : tuple of ndarrays
Indices of the maxima in arrays of integers. `extrema[k]` is the array of indices of axis *k* of *data*. Note that the return value is a tuple even when *data* is one-dimensional.

See also:

argrelmin, *argrelmax*

Notes

New in version 0.11.0.

Examples

```
>>> from scipy.signal import argrelextrema
>>> x = np.array([2, 1, 2, 3, 2, 0, 1, 0])
>>> argrelextrema(x, np.greater)
(array([3, 6]),)
>>> y = np.array([[1, 2, 1, 2],
...               [2, 2, 0, 0],
...               [5, 3, 4, 4]])
...
>>> argrelextrema(y, np.less, axis=1)
(array([0, 2]), array([2, 1]))
```

5.20.13 Spectral Analysis

periodogram(x[, fs, window, nfft, detrend, ...])	Estimate power spectral density using a periodogram.
welch(x[, fs, window, nperseg, noverlap, ...])	Estimate power spectral density using Welch's method.
csd(x, y[, fs, window, nperseg, noverlap, ...])	Estimate the cross power spectral density, Pxy, using Welch's method.
coherence(x, y[, fs, window, nperseg, ...])	Estimate the magnitude squared coherence estimate, Cxy, of discrete-time signals X and Y using Welch's method.
spectrogram(x[, fs, window, nperseg, ...])	Compute a spectrogram with consecutive Fourier transforms.
lombscargle(x, y, freqs)	Computes the Lomb-Scargle periodogram.
vectorstrength(events, period)	Determine the vector strength of the events corresponding to the given period.
stft(x[, fs, window, nperseg, noverlap, ...])	Compute the Short Time Fourier Transform (STFT).
istft(Zxx[, fs, window, nperseg, noverlap, ...])	Perform the inverse Short Time Fourier transform (iSTFT).
check_COLA(window, nperseg, noverlap[, tol])	Check whether the Constant OverLap Add (COLA) constraint is met

scipy.signal.**periodogram**(*x*, *fs=1.0*, *window='boxcar'*, *nfft=None*, *detrend='constant'*, *return_onesided=True*, *scaling='density'*, *axis=-1*)

Estimate power spectral density using a periodogram.

Parameters **x** : array_like

Time series of measurement values

fs : float, optional

Sampling frequency of the *x* time series. Defaults to 1.0.

window : str or tuple or array_like, optional

Desired window to use. See *get_window* for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be nperseg. Defaults to 'boxcar'.

nfft : int, optional

Length of the FFT used. If *None* the length of *x* will be used.

detrend : str or function or *False*, optional

Specifies how to detrend each segment. If *detrend* is a string, it is passed as the *type* argument to the *detrend* function. If it is a function, it takes

a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to 'constant'.

return_onesided : bool, optional
If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Note that for complex data, a two-sided spectrum is always returned.

scaling : { 'density', 'spectrum' }, optional
Selects between computing the power spectral density ('density') where *Pxx* has units of V**2/Hz and computing the power spectrum ('spectrum') where *Pxx* has units of V**2, if *x* is measured in V and *fs* is measured in Hz. Defaults to 'density'

axis : int, optional
Axis along which the periodogram is computed; the default is over the last axis (i.e. `axis=-1`).

Returns

f : ndarray
Array of sample frequencies.

Pxx : ndarray
Power spectral density or power spectrum of *x*.

See also:

`welch` Estimate power spectral density using Welch's method

`lombscargle` Lomb-Scargle periodogram for unevenly sampled data

Notes

New in version 0.12.0.

Examples

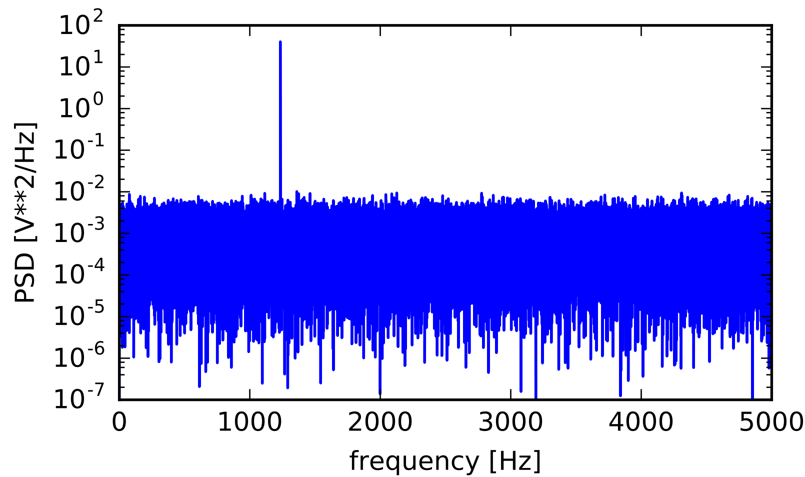
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> np.random.seed(1234)
```

Generate a test signal, a 2 Vrms sine wave at 1234 Hz, corrupted by 0.001 V**2/Hz of white noise sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
```

Compute and plot the power spectral density.

```
>>> f, Pxx_den = signal.periodogram(x, fs)
>>> plt.semilogy(f, Pxx_den)
>>> plt.ylim([1e-7, 1e2])
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD [V**2/Hz]')
>>> plt.show()
```

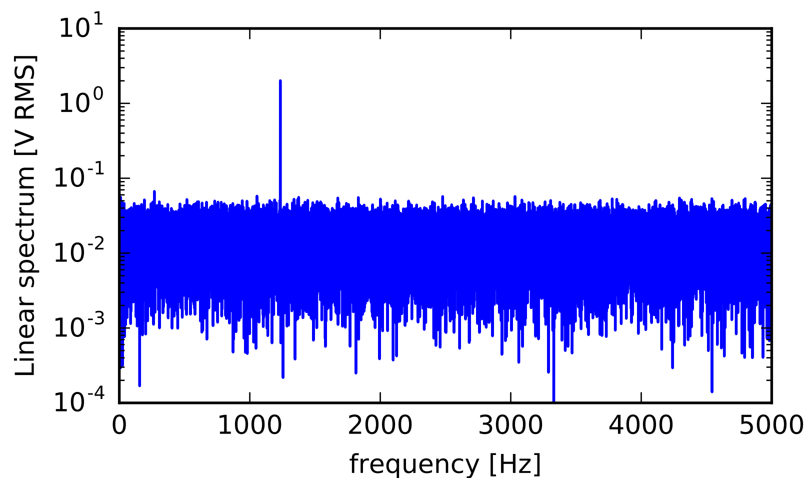


If we average the last half of the spectral density, to exclude the peak, we can recover the noise power on the signal.

```
>>> np.mean(Pxx_den[256:])
0.0018156616014838548
```

Now compute and plot the power spectrum.

```
>>> f, Pxx_spec = signal.periodogram(x, fs, 'flattop', scaling='spectrum')
>>> plt.figure()
>>> plt.semilogy(f, np.sqrt(Pxx_spec))
>>> plt.ylim([1e-4, 1e1])
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('Linear spectrum [V RMS]')
>>> plt.show()
```



The peak height in the power spectrum is an estimate of the RMS amplitude.

```
>>> np.sqrt(Pxx_spec.max())
2.0077340678640727
```

`scipy.signal.welch`(*x*, *fs=1.0*, *window='hann'*, *nperseg=None*, *noverlap=None*, *nfft=None*, *detrend='constant'*, *return_onesided=True*, *scaling='density'*, *axis=-1*)

Estimate power spectral density using Welch's method.

Welch's method *[R270]* computes an estimate of the power spectral density by dividing the data into overlapping segments, computing a modified periodogram for each segment and averaging the periodograms.

Parameters

x : array_like
: Time series of measurement values

fs : float, optional
: Sampling frequency of the *x* time series. Defaults to 1.0.

window : str or tuple or array_like, optional
: Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be nperseg. Defaults to a Hann window.

nperseg : int, optional
: Length of each segment. Defaults to None, but if window is str or tuple, is set to 256, and if window is array_like, is set to the length of the window.

noverlap : int, optional
: Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*.

nfft : int, optional
: Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.

detrend : str or function or *False*, optional
: Specifies how to detrend each segment. If `detrend` is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to 'constant'.

return_onesided : bool, optional
: If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Note that for complex data, a two-sided spectrum is always returned.

scaling : { 'density', 'spectrum' }, optional
: Selects between computing the power spectral density ('density') where *Pxx* has units of V**2/Hz and computing the power spectrum ('spectrum') where *Pxx* has units of V**2, if *x* is measured in V and *fs* is measured in Hz. Defaults to 'density'

axis : int, optional
: Axis along which the periodogram is computed; the default is over the last axis (i.e. `axis=-1`).

Returns

f : ndarray
: Array of sample frequencies.

Pxx : ndarray
: Power spectral density or power spectrum of x.

See also:

`periodogram`
: Simple, optionally modified periodogram

`lombscargle`
: Lomb-Scargle periodogram for unevenly sampled data

Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default 'hann' window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap.

If *noverlap* is 0, this method is equivalent to Bartlett's method *[R271]*.

New in version 0.12.0.

References

[R270], *[R271]*

Examples

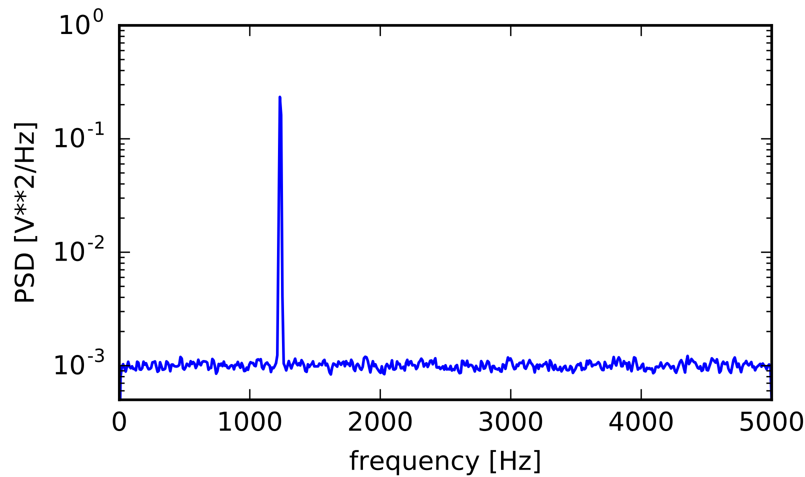
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> np.random.seed(1234)
```

Generate a test signal, a 2 Vrms sine wave at 1234 Hz, corrupted by 0.001 V**2/Hz of white noise sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2*np.sqrt(2)
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> x = amp*np.sin(2*np.pi*freq*time)
>>> x += np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
```

Compute and plot the power spectral density.

```
>>> f, Pxx_den = signal.welch(x, fs, nperseg=1024)
>>> plt.semilogy(f, Pxx_den)
>>> plt.ylim([0.5e-3, 1])
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('PSD [V**2/Hz]')
>>> plt.show()
```

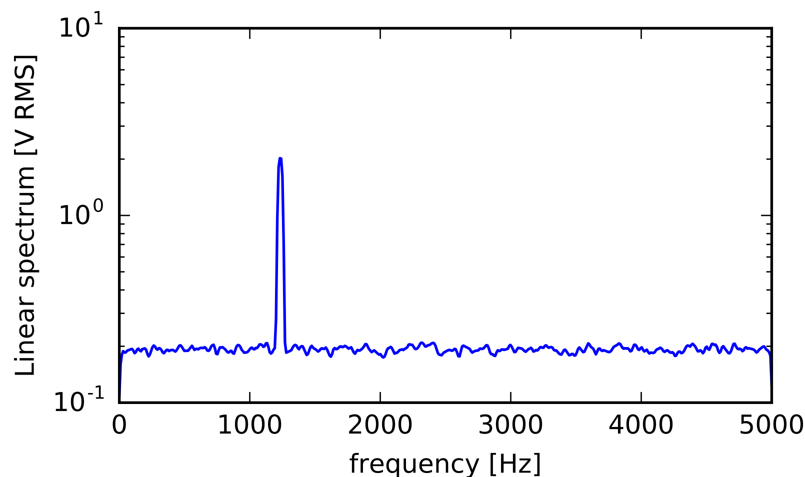


If we average the last half of the spectral density, to exclude the peak, we can recover the noise power on the signal.

```
>>> np.mean(Pxx_den[256:])
0.0009924865443739191
```

Now compute and plot the power spectrum.

```
>>> f, Pxx_spec = signal.welch(x, fs, 'flattop', 1024, scaling='spectrum')
>>> plt.figure()
>>> plt.semilogy(f, np.sqrt(Pxx_spec))
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('Linear spectrum [V RMS]')
>>> plt.show()
```



The peak height in the power spectrum is an estimate of the RMS amplitude.

```
>>> np.sqrt(Pxx_spec.max())
2.0077340678640727
```

scipy.signal.**csd**(*x*, *y*, *fs=1.0*, *window='hann'*, *nperseg=None*, *noverlap=None*, *nfft=None*, *detrend='constant'*, *return_onesided=True*, *scaling='density'*, *axis=-1*)

Estimate the cross power spectral density, Pxy, using Welch's method.

Parameters

x : array_like
: Time series of measurement values

y : array_like
: Time series of measurement values

fs : float, optional
: Sampling frequency of the *x* and *y* time series. Defaults to 1.0.

window : str or tuple or array_like, optional
: Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be nperseg. Defaults to a Hann window.

nperseg : int, optional
: Length of each segment. Defaults to None, but if window is str or tuple, is set to 256, and if window is array_like, is set to the length of the window.

noverlap: int, optional
: Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*.

nfft : int, optional
: Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.

detrend : str or function or *False*, optional
: Specifies how to detrend each segment. If `detrend` is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to 'constant'.

return_onesided : bool, optional
: If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Note that for complex data, a two-sided spectrum is always returned.

scaling : { 'density', 'spectrum' }, optional
: Selects between computing the cross spectral density ('density') where *Pxy* has units of V**2/Hz and computing the cross spectrum ('spectrum') where *Pxy* has units of V**2, if *x* and *y* are measured in V and *fs* is measured in Hz. Defaults to 'density'

axis : int, optional
: Axis along which the CSD is computed for both inputs; the default is over the last axis (i.e. `axis=-1`).

Returns

f : ndarray
: Array of sample frequencies.

Pxy : ndarray
: Cross spectral density or cross power spectrum of x,y.

See also:

`periodogram`
: Simple, optionally modified periodogram

`lombscargle`
: Lomb-Scargle periodogram for unevenly sampled data

`welch` Power spectral density by Welch's method. [Equivalent to csd(x,x)]

`coherence` Magnitude squared coherence by Welch's method.

Notes

By convention, Pxy is computed with the conjugate FFT of X multiplied by the FFT of Y.

If the input series differ in length, the shorter series will be zero-padded to match.

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default 'hann' window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap.

New in version 0.16.0.

References

[R213], [R214]

Examples

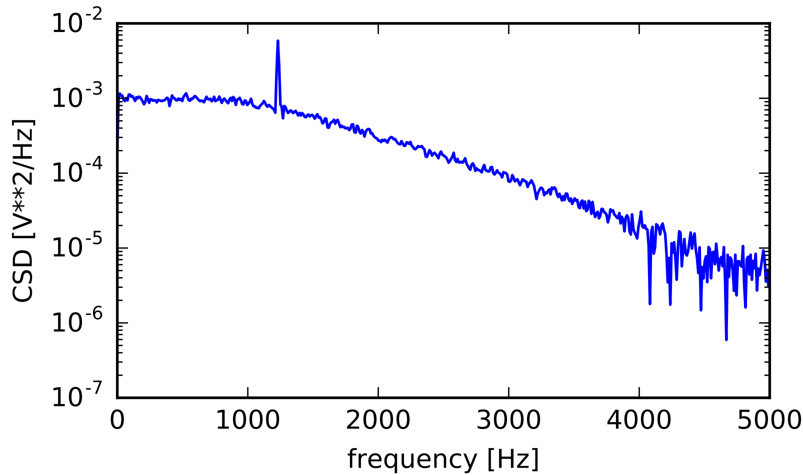
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Generate two test signals with some common features.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 20
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> b, a = signal.butter(2, 0.25, 'low')
>>> x = np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
>>> y = signal.lfilter(b, a, x)
>>> x += amp*np.sin(2*np.pi*freq*time)
>>> y += np.random.normal(scale=0.1*np.sqrt(noise_power), size=time.shape)
```

Compute and plot the magnitude of the cross spectral density.

```
>>> f, Pxy = signal.csd(x, y, fs, nperseg=1024)
>>> plt.semilogy(f, np.abs(Pxy))
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('CSD [V**2/Hz]')
>>> plt.show()
```



`scipy.signal.coherence`(*x*, *y*, *fs=1.0*, *window='hann'*, *nperseg=None*, *noverlap=None*, *nfft=None*, *detrend='constant'*, *axis=-1*)

Estimate the magnitude squared coherence estimate, Cxy, of discrete-time signals X and Y using Welch's method.

`Cxy = abs(Pxy)**2/(Pxx*Pyy)`, where *Pxx* and *Pyy* are power spectral density estimates of X and Y, and *Pxy* is the cross spectral density estimate of X and Y.

Parameters

x : array_like
: Time series of measurement values

y : array_like
: Time series of measurement values

fs : float, optional
: Sampling frequency of the *x* and *y* time series. Defaults to 1.0.

window : str or tuple or array_like, optional
: Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window.

nperseg : int, optional
: Length of each segment. Defaults to None, but if window is str or tuple, is set to 256, and if window is array_like, is set to the length of the window.

noverlap: int, optional
: Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*.

nfft : int, optional
: Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.

detrend : str or function or *False*, optional
: Specifies how to detrend each segment. If `detrend` is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to 'constant'.

axis : int, optional
: Axis along which the coherence is computed for both inputs; the default is over the last axis (i.e. `axis=-1`).

Returns

f : ndarray
: Array of sample frequencies.

Cxy : ndarray
Magnitude squared coherence of x and y.

See also:

periodogram
Simple, optionally modified periodogram

lombscargle
Lomb-Scargle periodogram for unevenly sampled data

welch Power spectral density by Welch's method.

csd Cross spectral density by Welch's method.

Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. For the default 'hann' window an overlap of 50% is a reasonable trade off between accurately estimating the signal power, while not over counting any of the data. Narrower windows may require a larger overlap.

New in version 0.16.0.

References

[R208], *[R209]*

Examples

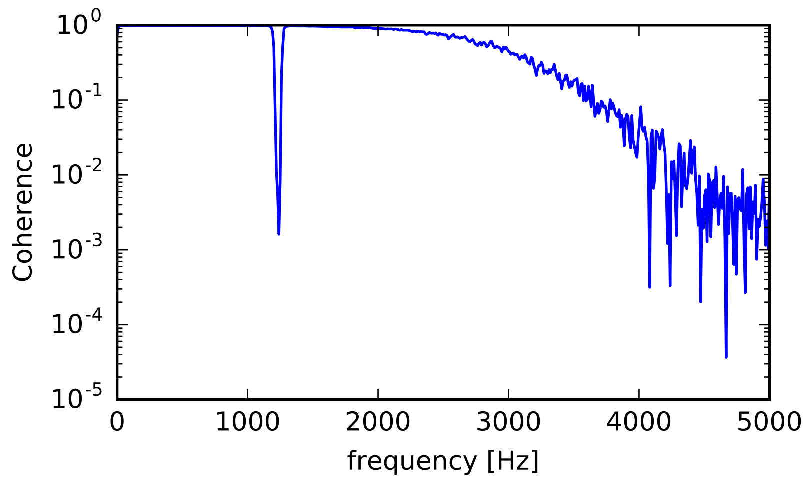
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Generate two test signals with some common features.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 20
>>> freq = 1234.0
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / fs
>>> b, a = signal.butter(2, 0.25, 'low')
>>> x = np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
>>> y = signal.lfilter(b, a, x)
>>> x += amp*np.sin(2*np.pi*freq*time)
>>> y += np.random.normal(scale=0.1*np.sqrt(noise_power), size=time.shape)
```

Compute and plot the coherence.

```
>>> f, Cxy = signal.coherence(x, y, fs, nperseg=1024)
>>> plt.semilogy(f, Cxy)
>>> plt.xlabel('frequency [Hz]')
>>> plt.ylabel('Coherence')
>>> plt.show()
```



`scipy.signal.spectrogram`(*x*, *fs=1.0*, *window=('tukey', 0.25)*, *nperseg=None*, *noverlap=None*, *nfft=None*, *detrend='constant'*, *return_onesided=True*, *scaling='density'*, *axis=-1*, *mode='psd'*)

Compute a spectrogram with consecutive Fourier transforms.

Spectrograms can be used as a way of visualizing the change of a nonstationary signal's frequency content over time.

Parameters

x : array_like
: Time series of measurement values

fs : float, optional
: Sampling frequency of the *x* time series. Defaults to 1.0.

window : str or tuple or array_like, optional
: Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be nperseg. Defaults to a Tukey window with shape parameter of 0.25.

nperseg : int, optional
: Length of each segment. Defaults to None, but if window is str or tuple, is set to 256, and if window is array_like, is set to the length of the window.

noverlap : int, optional
: Number of points to overlap between segments. If *None*, `noverlap = nperseg // 8`. Defaults to *None*.

nfft : int, optional
: Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.

detrend : str or function or *False*, optional
: Specifies how to detrend each segment. If `detrend` is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to 'constant'.

return_onesided : bool, optional
: If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Note that for complex data, a two-sided spectrum is always returned.

scaling : { 'density', 'spectrum' }, optional

Selects between computing the power spectral density ('density') where *Sxx* has units of V**2/Hz and computing the power spectrum ('spectrum') where *Sxx* has units of V**2, if *x* is measured in V and *fs* is measured in Hz. Defaults to 'density'.

axis : int, optional

Axis along which the spectrogram is computed; the default is over the last axis (i.e. `axis=-1`).

mode : str, optional

Defines what kind of return values are expected. Options are ['psd', 'complex', 'magnitude', 'angle', 'phase']. 'complex' is equivalent to the output of `stft` with no padding or boundary extension. 'magnitude' returns the absolute magnitude of the STFT. 'angle' and 'phase' return the complex angle of the STFT, with and without unwrapping, respectively.

Returns

f : ndarray

Array of sample frequencies.

t : ndarray

Array of segment times.

Sxx : ndarray

Spectrogram of x. By default, the last axis of Sxx corresponds to the segment times.

See also:

`periodogram` Simple, optionally modified periodogram

`lombscargle` Lomb-Scargle periodogram for unevenly sampled data

`welch` Power spectral density by Welch's method.

`csd` Cross spectral density by Welch's method.

Notes

An appropriate amount of overlap will depend on the choice of window and on your requirements. In contrast to welch's method, where the entire data stream is averaged over, one may wish to use a smaller overlap (or perhaps none at all) when computing a spectrogram, to maintain some statistical independence between individual segments. It is for this reason that the default window is a Tukey window with 1/8th of a window's length overlap at each end.

New in version 0.16.0.

References

[R264]

Examples

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

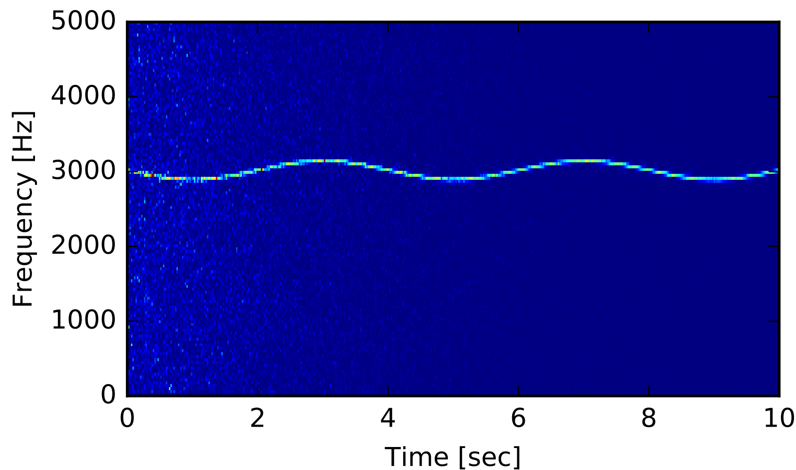
Generate a test signal, a 2 Vrms sine wave whose frequency is slowly modulated around 3kHz, corrupted by white noise of exponentially decreasing magnitude sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2 * np.sqrt(2)
>>> noise_power = 0.01 * fs / 2
>>> time = np.arange(N) / float(fs)
>>> mod = 500*np.cos(2*np.pi*0.25*time)
```

```
>>> carrier = amp * np.sin(2*np.pi*3e3*time + mod)
>>> noise = np.random.normal(scale=np.sqrt(noise_power), size=time.shape)
>>> noise *= np.exp(-time/5)
>>> x = carrier + noise
```

Compute and plot the spectrogram.

```
>>> f, t, Sxx = signal.spectrogram(x, fs)
>>> plt.pcolormesh(t, f, Sxx)
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```



`scipy.signal.`**`lombscargle`**(*x*, *y*, *freqs*)

Computes the Lomb-Scargle periodogram.

The Lomb-Scargle periodogram was developed by Lomb *[R247]* and further extended by Scargle *[R248]* to find, and test the significance of weak periodic signals with uneven temporal sampling.

The computed periodogram is unnormalized, it takes the value `(A**2) * N/4` for a harmonic signal with amplitude A for sufficiently large N.

Parameters

x : array_like
  Sample times.

y : array_like
  Measurement values.

freqs : array_like
  Angular frequencies for output periodogram.

Returns

pgram : array_like
  Lomb-Scargle periodogram.

Raises

ValueError
  If the input arrays *x* and *y* do not have the same shape.

Notes

This subroutine calculates the periodogram using a slightly modified algorithm due to Townsend *[R249]* which allows the periodogram to be calculated using only a single pass through the input arrays for each frequency.

The algorithm running time scales roughly as O(x * freqs) or O(N^2) for a large number of samples and frequencies.

References

[R247], [R248], [R249]

Examples

```
>>> import scipy.signal
>>> import matplotlib.pyplot as plt
```

First define some input parameters for the signal:

```
>>> A = 2.
>>> w = 1.
>>> phi = 0.5 * np.pi
>>> nin = 1000
>>> nout = 100000
>>> frac_points = 0.9 # Fraction of points to select
```

Randomly select a fraction of an array with timesteps:

```
>>> r = np.random.rand(nin)
>>> x = np.linspace(0.01, 10*np.pi, nin)
>>> x = x[r >= frac_points]
>>> normval = x.shape[0] # For normalization of the periodogram
```

Plot a sine wave for the selected times:

```
>>> y = A * np.sin(w*x+phi)
```

Define the array of frequencies for which to compute the periodogram:

```
>>> f = np.linspace(0.01, 10, nout)
```

Calculate Lomb-Scargle periodogram:

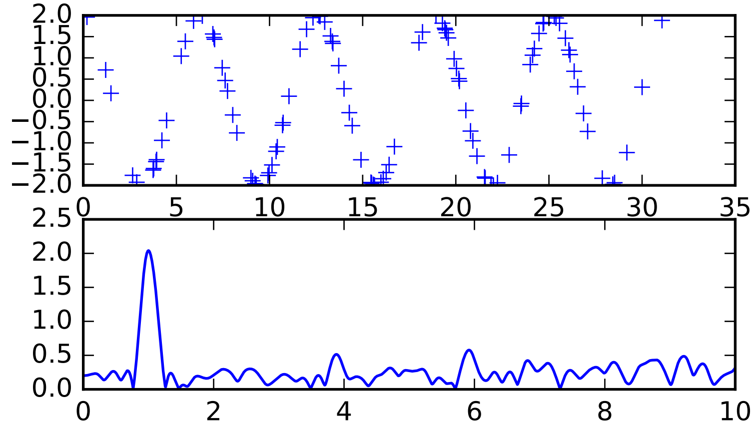
```
>>> import scipy.signal as signal
>>> pgram = signal.lombscargle(x, y, f)
```

Now make a plot of the input data:

```
>>> plt.subplot(2, 1, 1)
<matplotlib.axes.AxesSubplot object at 0x102154f50>
>>> plt.plot(x, y, 'b+')
[<matplotlib.lines.Line2D object at 0x102154a10>]
```

Then plot the normalized periodogram:

```
>>> plt.subplot(2, 1, 2)
<matplotlib.axes.AxesSubplot object at 0x104b0a990>
>>> plt.plot(f, np.sqrt(4*(pgram/normval)))
[<matplotlib.lines.Line2D object at 0x104b2f910>]
>>> plt.show()
```



`scipy.signal.`**`vectorstrength`**(*events*, *period*)

Determine the vector strength of the events corresponding to the given period.

The vector strength is a measure of phase synchrony, how well the timing of the events is synchronized to a single period of a periodic signal.

If multiple periods are used, calculate the vector strength of each. This is called the "resonating vector strength".

Parameters **events** : 1D array_like

An array of time points containing the timing of the events.

period : float or array_like

The period of the signal that the events should synchronize to. The period is in the same units as *events*. It can also be an array of periods, in which case the outputs are arrays of the same length.

Returns **strength** : float or 1D array

The strength of the synchronization. 1.0 is perfect synchronization and 0.0 is no synchronization. If *period* is an array, this is also an array with each element containing the vector strength at the corresponding period.

phase : float or array

The phase that the events are most strongly synchronized to in radians. If *period* is an array, this is also an array with each element containing the phase for the corresponding period.

References

van Hemmen, JL, Longtin, A, and Vollmayr, AN. Testing resonating vector
strength: Auditory system, electric fish, and noise. Chaos 21, 047508 (2011); DOI:10.1063/1.3670512.

van Hemmen, JL. Vector strength after Goldberg, Brown, and von Mises:
biological and mathematical perspectives. Biol Cybern. 2013 Aug;107(4):385-96. DOI:10.1007/s00422-013-0561-7.

van Hemmen, JL and Vollmayr, AN. Resonating vector strength: what happens
when we vary the "probing" frequency while keeping the spike times fixed. Biol Cybern. 2013 Aug;107(4):491-94. DOI:10.1007/s00422-013-0560-8.

`scipy.signal.`**`stft`**(*x*, *fs=1.0*, *window='hann'*, *nperseg=256*, *noverlap=None*, *nfft=None*, *detrend=False*, *return_onesided=True*, *boundary='zeros'*, *padded=True*, *axis=-1*)

Compute the Short Time Fourier Transform (STFT).

STFTs can be used as a way of quantifying the change of a nonstationary signal's frequency and phase content over time.

Parameters

x : array_like
    Time series of measurement values

fs : float, optional
    Sampling frequency of the *x* time series. Defaults to 1.0.

window : str or tuple or array_like, optional
    Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be nperseg. Defaults to a Hann window.

nperseg : int, optional
    Length of each segment. Defaults to 256.

noverlap : int, optional
    Number of points to overlap between segments. If *None*, `noverlap = nperseg // 2`. Defaults to *None*. When specified, the COLA constraint must be met (see Notes below).

nfft : int, optional
    Length of the FFT used, if a zero padded FFT is desired. If *None*, the FFT length is *nperseg*. Defaults to *None*.

detrend : str or function or *False*, optional
    Specifies how to detrend each segment. If `detrend` is a string, it is passed as the *type* argument to the `detrend` function. If it is a function, it takes a segment and returns a detrended segment. If `detrend` is *False*, no detrending is done. Defaults to *False*.

return_onesided : bool, optional
    If *True*, return a one-sided spectrum for real data. If *False* return a two-sided spectrum. Note that for complex data, a two-sided spectrum is always returned. Defaults to *True*.

boundary : str or None, optional
    Specifies whether the input signal is extended at both ends, and how to generate the new values, in order to center the first windowed segment on the first input point. This has the benefit of enabling reconstruction of the first input point when the employed window function starts at zero. Valid options are `['even', 'odd', 'constant', 'zeros', None]`. Defaults to 'zeros', for zero padding extension. I.e. `[1, 2, 3, 4]` is extended to `[0, 1, 2, 3, 4, 0]` for `nperseg=3`.

padded : bool, optional
    Specifies whether the input signal is zero-padded at the end to make the signal fit exactly into an integer number of window segments, so that all of the signal is included in the output. Defaults to *True*. Padding occurs after boundary extension, if *boundary* is not *None*, and *padded* is *True*, as is the default.

axis : int, optional
    Axis along which the STFT is computed; the default is over the last axis (i.e. `axis=-1`).

Returns

f : ndarray
    Array of sample frequencies.

t : ndarray
    Array of segment times.

Zxx : ndarray
    STFT of *x*. By default, the last axis of *Zxx* corresponds to the segment times.

See also:

istft Inverse Short Time Fourier Transform
check_COLA Check whether the Constant OverLap Add (COLA) constraint is met
welch Power spectral density by Welch's method.
spectrogram Spectrogram by Welch's method.
csd Cross spectral density by Welch's method.
lombscargle Lomb-Scargle periodogram for unevenly sampled data

Notes

In order to enable inversion of an STFT via the inverse STFT in `istft`, the signal windowing must obey the constraint of "Constant OverLap Add" (COLA), and the input signal must have complete windowing coverage (i.e. `(x.shape[axis] - nperseg) % (nperseg-noverlap) == 0`). The *padded* argument may be used to accomplish this.

The COLA constraint ensures that every point in the input data is equally weighted, thereby avoiding aliasing and allowing full reconstruction. Whether a choice of *window*, *nperseg*, and *noverlap* satisfy this constraint can be tested with `check_COLA`.

New in version 0.19.0.

References

[R265], [R266]

Examples

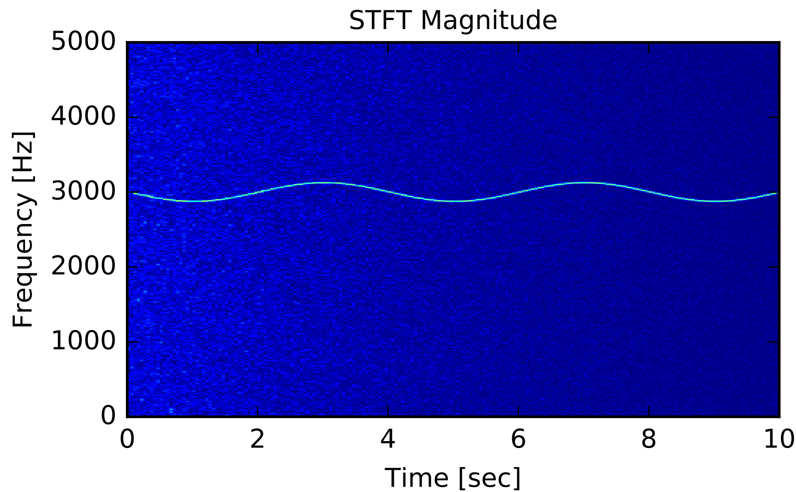
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Generate a test signal, a 2 Vrms sine wave whose frequency is slowly modulated around 3kHz, corrupted by white noise of exponentially decreasing magnitude sampled at 10 kHz.

```
>>> fs = 10e3
>>> N = 1e5
>>> amp = 2 * np.sqrt(2)
>>> noise_power = 0.01 * fs / 2
>>> time = np.arange(N) / float(fs)
>>> mod = 500*np.cos(2*np.pi*0.25*time)
>>> carrier = amp * np.sin(2*np.pi*3e3*time + mod)
>>> noise = np.random.normal(scale=np.sqrt(noise_power),
...                          size=time.shape)
>>> noise *= np.exp(-time/5)
>>> x = carrier + noise
```

Compute and plot the STFT's magnitude.

```
>>> f, t, Zxx = signal.stft(x, fs, nperseg=1000)
>>> plt.pcolormesh(t, f, np.abs(Zxx), vmin=0, vmax=amp)
>>> plt.title('STFT Magnitude')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.show()
```



`scipy.signal.istft`(*Zxx*, *fs=1.0*, *window='hann'*, *nperseg=None*, *noverlap=None*, *nfft=None*, *input_onesided=True*, *boundary=True*, *time_axis=-1*, *freq_axis=-2*)

Perform the inverse Short Time Fourier transform (iSTFT).

Parameters

Zxx : array_like

STFT of the signal to be reconstructed. If a purely real array is passed, it will be cast to a complex data type.

fs : float, optional

Sampling frequency of the time series. Defaults to 1.0.

window : str or tuple or array_like, optional

Desired window to use. See *get_window* for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be *nperseg*. Defaults to a Hann window. Must match the window used to generate the STFT for faithful inversion.

nperseg : int, optional

Number of data points corresponding to each STFT segment. This parameter must be specified if the number of data points per segment is odd, or if the STFT was padded via `nfft > nperseg`. If *None*, the value depends on the shape of *Zxx* and *input_onesided*. If *input_onesided* is True, `nperseg=2*(Zxx.shape[freq_axis] - 1)`. Otherwise, `nperseg=Zxx.shape[freq_axis]`. Defaults to *None*.

noverlap : int, optional

Number of points to overlap between segments. If *None*, half of the segment length. Defaults to *None*. When specified, the COLA constraint must be met (see Notes below), and should match the parameter used to generate the STFT. Defaults to *None*.

nfft : int, optional

Number of FFT points corresponding to each STFT segment. This parameter must be specified if the STFT was padded via `nfft > nperseg`. If *None*, the default values are the same as for *nperseg*, detailed above, with one exception: if *input_onesided* is True and `nperseg==2*Zxx.shape[freq_axis] - 1`, *nfft* also takes on that value. This case allows the proper inversion of an odd-length unpadded STFT using `nfft=None`. Defaults to *None*.

input_onesided : bool, optional

If *True*, interpret the input array as one-sided FFTs, such as is returned by `stft` with `return_onesided=True` and `numpy.fft.rfft`. If *False*, interpret the input as a a two-sided FFT. Defaults to *True*.

boundary : bool, optional

Specifies whether the input signal was extended at its boundaries by supplying a non-*None* `boundary` argument to `stft`. Defaults to *True*.

time_axis : int, optional

Where the time segments of the STFT is located; the default is the last axis (i.e. `axis=-1`).

freq_axis : int, optional

Where the frequency axis of the STFT is located; the default is the penultimate axis (i.e. `axis=-2`).

Returns

t : ndarray

Array of output data times.

x : ndarray

iSTFT of *Zxx*.

See also:

`stft` Short Time Fourier Transform

`check_COLA` Check whether the Constant OverLap Add (COLA) constraint is met

Notes

In order to enable inversion of an STFT via the inverse STFT with `istft`, the signal windowing must obey the constraint of "Constant OverLap Add" (COLA). This ensures that every point in the input data is equally weighted, thereby avoiding aliasing and allowing full reconstruction. Whether a choice of *window*, *nperseg*, and *noverlap* satisfy this constraint can be tested with `check_COLA`, by using `nperseg = Zxx.shape[freq_axis]`.

An STFT which has been modified (via masking or otherwise) is not guaranteed to correspond to a exactly realizible signal. This function implements the iSTFT via the least-squares esimation algorithm detailed in *[R242]*, which produces a signal that minimizes the mean squared error between the STFT of the returned signal and the modified STFT.

New in version 0.19.0.

References

[R241], *[R242]*

Examples

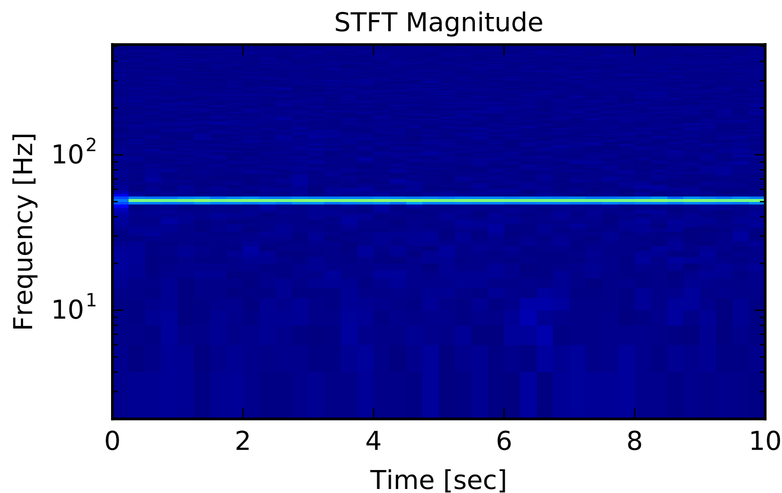
```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
```

Generate a test signal, a 2 Vrms sine wave at 50Hz corrupted by 0.001 V**2/Hz of white noise sampled at 1024 Hz.

```
>>> fs = 1024
>>> N = 10*fs
>>> nperseg = 512
>>> amp = 2 * np.sqrt(2)
>>> noise_power = 0.001 * fs / 2
>>> time = np.arange(N) / float(fs)
>>> carrier = amp * np.sin(2*np.pi*50*time)
>>> noise = np.random.normal(scale=np.sqrt(noise_power),
...                          size=time.shape)
>>> x = carrier + noise
```

Compute the STFT, and plot its magnitude

```
>>> f, t, Zxx = signal.stft(x, fs=fs, nperseg=nperseg)
>>> plt.figure()
>>> plt.pcolormesh(t, f, np.abs(Zxx), vmin=0, vmax=amp)
>>> plt.ylim([f[1], f[-1]])
>>> plt.title('STFT Magnitude')
>>> plt.ylabel('Frequency [Hz]')
>>> plt.xlabel('Time [sec]')
>>> plt.yscale('log')
>>> plt.show()
```

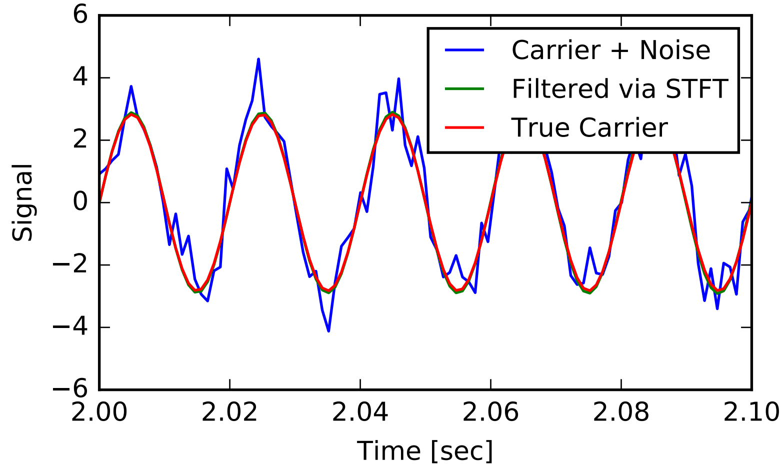


Zero the components that are 10% or less of the carrier magnitude, then convert back to a time series via inverse STFT

```
>>> Zxx = np.where(np.abs(Zxx) >= amp/10, Zxx, 0)
>>> _, xrec = signal.istft(Zxx, fs)
```

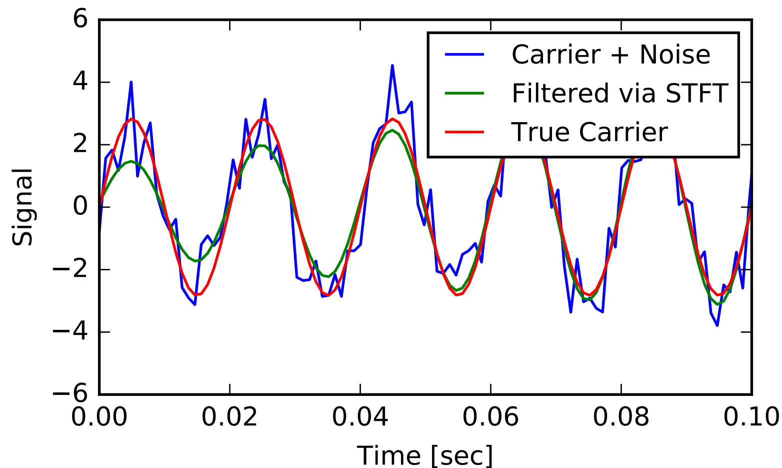
Compare the cleaned signal with the original and true carrier signals.

```
>>> plt.figure()
>>> plt.plot(time, x, time, xrec, time, carrier)
>>> plt.xlim([2, 2.1])
>>> plt.xlabel('Time [sec]')
>>> plt.ylabel('Signal')
>>> plt.legend(['Carrier + Noise', 'Filtered via STFT', 'True Carrier'])
>>> plt.show()
```



Note that the cleaned signal does not start as abruptly as the original, since some of the coefficients of the transient were also removed:

```
>>> plt.figure()
>>> plt.plot(time, x, time, xrec, time, carrier)
>>> plt.xlim([0, 0.1])
>>> plt.xlabel('Time [sec]')
>>> plt.ylabel('Signal')
>>> plt.legend(['Carrier + Noise', 'Filtered via STFT', 'True Carrier'])
>>> plt.show()
```



`scipy.signal.`**`check_COLA`**(*window*, *nperseg*, *noverlap*, *tol=1e-10*)

Check whether the Constant OverLap Add (COLA) constraint is met

Parameters **window** : str or tuple or array_like

Desired window to use. See `get_window` for a list of windows and required parameters. If *window* is array_like it will be used directly as the window and its length must be *nperseg*.

nperseg : int
    Length of each segment.

noverlap : int
    Number of points to overlap between segments.

tol : float, optional
    The allowed variance of a bin's weighted sum from the median bin sum.

Returns **verdict** : bool
    True if chosen combination satisfies COLA within *tol*, *False* otherwise

See also:

`stft`	Short Time Fourier Transform
`istft`	Inverse Short Time Fourier Transform

Notes

In order to enable inversion of an STFT via the inverse STFT in `istft`, the signal windowing must obey the constraint of "Constant OverLap Add" (COLA). This ensures that every point in the input data is equally weighted, thereby avoiding aliasing and allowing full reconstruction.

Some examples of windows that satisfy COLA:

- Rectangular window at overlap of 0, 1/2, 2/3, 3/4, ...
- Bartlett window at overlap of 1/2, 3/4, 5/6, ...
- Hann window at 1/2, 2/3, 3/4, ...
- Any Blackman family window at 2/3 overlap
- Any window with `noverlap = nperseg-1`

A very comprehensive list of other windows may be found in *[R207]*, wherein the COLA condition is satisfied when the "Amplitude Flatness" is unity.

New in version 0.19.0.

References

[R206], *[R207]*

Examples

```
>>> from scipy import signal
```

Confirm COLA condition for rectangular window of 75% (3/4) overlap:

```
>>> signal.check_COLA(signal.boxcar(100), 100, 75)
True
```

COLA is not true for 25% (1/4) overlap, though:

```
>>> signal.check_COLA(signal.boxcar(100), 100, 25)
False
```

"Symmetrical" Hann window (for filter design) is not COLA:

```
>>> signal.check_COLA(signal.hann(120, sym=True), 120, 60)
False
```

"Periodic" or "DFT-even" Hann window (for FFT analysis) is COLA for overlap of 1/2, 2/3, 3/4, etc.:

```
>>> signal.check_COLA(signal.hann(120, sym=False), 120, 60)
True
```

```
>>> signal.check_COLA(signal.hann(120, sym=False), 120, 80)
True
```

```
>>> signal.check_COLA(signal.hann(120, sym=False), 120, 90)
True
```

5.21 Sparse matrices (`scipy.sparse`)

SciPy 2-D sparse matrix package for numeric data.

5.21.1 Contents

Sparse matrix classes

bsr_matrix(arg1[, shape, dtype, copy, blocksize])	Block Sparse Row matrix
coo_matrix(arg1[, shape, dtype, copy])	A sparse matrix in COOrdinate format.
csc_matrix(arg1[, shape, dtype, copy])	Compressed Sparse Column matrix
csr_matrix(arg1[, shape, dtype, copy])	Compressed Sparse Row matrix
dia_matrix(arg1[, shape, dtype, copy])	Sparse matrix with DIAgonal storage
dok_matrix(arg1[, shape, dtype, copy])	Dictionary Of Keys based sparse matrix.
lil_matrix(arg1[, shape, dtype, copy])	Row-based linked list sparse matrix
spmatrix([maxprint])	This class provides a base class for all sparse matrices.

class `scipy.sparse.`**`bsr_matrix`**(*arg1*, *shape=None*, *dtype=None*, *copy=False*, *blocksize=None*)

Block Sparse Row matrix

This can be instantiated in several ways:

bsr_matrix(D, [blocksize=(R,C)])
where D is a dense matrix or 2-D ndarray.

bsr_matrix(S, [blocksize=(R,C)])
with another sparse matrix S (equivalent to S.tobsr())

bsr_matrix((M, N), [blocksize=(R,C), dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

bsr_matrix((data, ij), [blocksize=(R,C), shape=(M, N)])
where `data` and `ij` satisfy `a[ij[0, k], ij[1, k]] = data[k]`

bsr_matrix((data, indices, indptr), [shape=(M, N)])
is the standard BSR representation where the block column indices for row i are stored in `indices[indptr[i]:indptr[i+1]]` and their corresponding block values are stored in `data[ indptr[i]: indptr[i+1] ]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Summary of BSR format

The Block Compressed Row (BSR) format is very similar to the Compressed Sparse Row (CSR) format. BSR is appropriate for sparse matrices with dense sub matrices like the last example below. Block matrices often arise in vector-valued finite element discretizations. In such cases, BSR is considerably more efficient than CSR and CSC for many sparse arithmetic operations.

Blocksize

The blocksize (R,C) must evenly divide the shape of the matrix (M,N). That is, R and C must satisfy the relationship `M % R = 0` and `N % C = 0`.

If no blocksize is specified, a simple heuristic is applied to determine an appropriate blocksize.

Examples

```
>>> from scipy.sparse import bsr_matrix
>>> bsr_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3 ,4, 5, 6])
>>> bsr_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

```
>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6]).repeat(4).reshape(6, 2, 2)
>>> bsr_matrix((data,indices,indptr), shape=(6, 6)).toarray()
array([[1, 1, 0, 0, 2, 2],
       [1, 1, 0, 0, 2, 2],
       [0, 0, 0, 0, 3, 3],
       [0, 0, 0, 0, 3, 3],
       [4, 4, 5, 5, 6, 6],
       [4, 4, 5, 5, 6, 6]])
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.
has_sorted_indices	Determine whether the matrix has sorted indices

`bsr_matrix.`**`shape`**
Get shape of a matrix.

`bsr_matrix.`**`nnz`**
Number of stored values, including explicit zeros.

See also:

`count_nonzero`
Number of non-zero entries

`bsr_matrix.`**`has_sorted_indices`**
Determine whether the matrix has sorted indices

Returns

- True: if the indices of the matrix are in sorted order
- False: otherwise

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	Data array of the matrix
indices	BSR format index array
indptr	BSR format index pointer array
blocksize	Block size of the matrix

Methods

arcsin()	Element-wise arcsin.
arcsinh()	Element-wise arcsinh.
arctan()	Element-wise arctan.
arctanh()	Element-wise arctanh.
argmax([axis, out])	Return indices of minimum elements along an axis.
argmin([axis, out])	Return indices of minimum elements along an axis.
asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
ceil()	Element-wise ceil.
check_format([full_check])	check whether the matrix format is valid
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
deg2rad()	Element-wise deg2rad.
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
eliminate_zeros()	Remove zero elements in-place.
expm1()	Element-wise expm1.
floor()	Element-wise floor.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).
log1p()	Element-wise log1p.
matmat(*args, **kwds)	*matmat* is deprecated!
matvec(*args, **kwds)	*matvec* is deprecated!
max([axis, out])	Return the maximum of the matrix or maximum along an axis.
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.

Continued on next page

Table 5.150 – continued from previous page

min([axis, out])	Return the minimum of the matrix or maximum along an axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix, vector, or scalar.
nonzero()	nonzero indices
power(n[, dtype])	This function performs element-wise power.
prune()	Remove empty space after all non-zero elements.
rad2deg()	Element-wise rad2deg.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
rint()	Element-wise rint.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sign()	Element-wise sign.
sin()	Element-wise sin.
sinh()	Element-wise sinh.
sort_indices()	Sort the indices of this matrix *in place*
sorted_indices()	Return a copy of this matrix with sorted indices
sqrt()	Element-wise sqrt.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
sum_duplicates()	Eliminate duplicate matrix entries by adding them together
tan()	Element-wise tan.
tanh()	Element-wise tanh.
toarray([order, out])	See the docstring for *spmatrix.toarray*.
tobsr([blocksize, copy])	Convert this matrix into Block Sparse Row Format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.
tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
trunc()	Element-wise trunc.

bsr_matrix.**arcsin**()

Element-wise arcsin.

See numpy.arcsin for more information.

bsr_matrix.**arcsinh**()

Element-wise arcsinh.

See numpy.arcsinh for more information.

bsr_matrix.**arctan**()

Element-wise arctan.

See numpy.arctan for more information.

bsr_matrix.**arctanh**()

Element-wise arctanh.

See numpy.arctanh for more information.

bsr_matrix.**argmax**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several maximum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int

Indices of maximum elements. If matrix, its size along *axis* is 1.

bsr_matrix.**argmin**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several minimum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int

Indices of minimum elements. If matrix, its size along *axis* is 1.

bsr_matrix.**asformat**(*format*)

Return this matrix in a given sparse format

Parameters **format** : {string, None}

desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

bsr_matrix.**asfptype**()

Upcast matrix to a floating point format (if necessary)

bsr_matrix.**astype**(*t*)

Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype

Typecode or data-type to which to cast the data.

bsr_matrix.**ceil**()
Element-wise ceil.

See numpy.ceil for more information.

bsr_matrix.**check_format**(*full_check=True*)
check whether the matrix format is valid

Parameters:

full_check: True - rigorous check, O(N) operations : default False - basic check, O(1) operations

bsr_matrix.**conj**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

bsr_matrix.**conjugate**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

bsr_matrix.**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

bsr_matrix.**count_nonzero**()
Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

bsr_matrix.**deg2rad**()
Element-wise deg2rad.

See numpy.deg2rad for more information.

bsr_matrix.**diagonal**()
Returns the main diagonal of the matrix

bsr_matrix.**dot**(*other*)
Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

bsr_matrix.**eliminate_zeros**()
Remove zero elements in-place.

bsr_matrix.**expm1**()
Element-wise expm1.

See numpy.expm1 for more information.

bsr_matrix.**floor**()
Element-wise floor.

See numpy.floor for more information.

bsr_matrix.**getH**()
Return the Hermitian transpose of this matrix.

See also:

np.matrix.getH
NumPy's implementation of *getH* for matrices

bsr_matrix.**get_shape**()
Get shape of a matrix.

bsr_matrix.**getcol**(*j*)
Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

bsr_matrix.**getformat**()
Format of a matrix representation as a string.

bsr_matrix.**getmaxprint**()
Maximum number of elements to display when printed.

bsr_matrix.**getnnz**(*axis=None*)
Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1
Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero
Number of non-zero entries

bsr_matrix.**getrow**(*i*)
Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).

bsr_matrix.**log1p**()
Element-wise log1p.

See numpy.log1p for more information.

bsr_matrix.**matmat**(**args*, ***kwds*)
matmat is deprecated! BSR matmat is deprecated in scipy 0.19.0. Use * operator instead.

Multiply this sparse matrix by other matrix.

bsr_matrix.**matvec**(**args*, ***kwds*)
matvec is deprecated! BSR matvec is deprecated in scipy 0.19.0. Use * operator instead.

Multiply matrix by vector.

bsr_matrix.**max**(*axis=None*, *out=None*)
Return the maximum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amax** : coo_matrix or scalar

Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

`min` The minimum value of a sparse matrix along a given axis.

`np.matrix.max`
NumPy's implementation of 'max' for matrices

`bsr_matrix.`**`maximum`**`(`*other*`)`
Element-wise maximum between this and another matrix.

`bsr_matrix.`**`mean`**`(`*axis=None*, *dtype=None*, *out=None*`)`
Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional

Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional

Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

`np.matrix.mean`
NumPy's implementation of 'mean' for matrices

`bsr_matrix.`**`min`**`(`*axis=None*, *out=None*`)`
Return the minimum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional

Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amin** : coo_matrix or scalar

Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

`max` The maximum value of a sparse matrix along a given axis.

`np.matrix.min`
NumPy's implementation of 'min' for matrices

`bsr_matrix.`**`minimum`**`(`*other*`)`
Element-wise minimum between this and another matrix.

`bsr_matrix.`**`multiply`**`(`*other*`)`
Point-wise multiplication by another matrix, vector, or scalar.

`bsr_matrix.`**`nonzero`**`()`
nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`bsr_matrix.`**`power`**`(`*n*, *dtype=None*`)`
This function performs element-wise power.

Parameters **n** : n is a scalar
dtype : If dtype is not specified, the current dtype will be preserved.

`bsr_matrix.`**`prune`**`()`
Remove empty space after all non-zero elements.

`bsr_matrix.`**`rad2deg`**`()`
Element-wise rad2deg.

See numpy.rad2deg for more information.

`bsr_matrix.`**`reshape`**`(`*shape*, *order='C'*`)`
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.
order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

`np.matrix.reshape`
NumPy's implementation of 'reshape' for matrices

`bsr_matrix.`**`rint`**`()`
Element-wise rint.

See numpy.rint for more information.

`bsr_matrix.`**`set_shape`**`(`*shape*`)`
See `reshape`.

`bsr_matrix.`**`setdiag`**`(`*values*, *k=0*`)`
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.

Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.

k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

`bsr_matrix.`**`sign`**`()`
Element-wise sign.

See numpy.sign for more information.

`bsr_matrix.`**`sin`**`()`
Element-wise sin.

See numpy.sin for more information.

`bsr_matrix.`**`sinh`**`()`
Element-wise sinh.

See numpy.sinh for more information.

`bsr_matrix.`**`sort_indices`**`()`
Sort the indices of this matrix *in place*

`bsr_matrix.`**`sorted_indices`**`()`
Return a copy of this matrix with sorted indices

`bsr_matrix.`**`sqrt`**`()`
Element-wise sqrt.

See numpy.sqrt for more information.

`bsr_matrix.`**`sum`**`(`*axis=None*, *dtype=None*, *out=None*`)`
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).

dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

`np.matrix.sum`
NumPy's implementation of 'sum' for matrices

`bsr_matrix.`**`sum_duplicates`**`()`
Eliminate duplicate matrix entries by adding them together

The is an *in place* operation

`bsr_matrix.`**`tan`**`()`

Element-wise tan.

See numpy.tan for more information.

`bsr_matrix.`**`tanh`**`()`

Element-wise tanh.

See numpy.tanh for more information.

`bsr_matrix.`**`toarray`**`(`*order=None*, *out=None*`)`

See the docstring for *`spmatrix.toarray`*.

`bsr_matrix.`**`tobsr`**`(`*blocksize=None*, *copy=False*`)`

Convert this matrix into Block Sparse Row Format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

If blocksize=(R, C) is provided, it will be used for determining block size of the bsr_matrix.

`bsr_matrix.`**`tocoo`**`(`*copy=True*`)`

Convert this matrix to COOrdinate format.

When copy=False the data array will be shared between this matrix and the resultant coo_matrix.

`bsr_matrix.`**`tocsc`**`(`*copy=False*`)`

Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

`bsr_matrix.`**`tocsr`**`(`*copy=False*`)`

Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

`bsr_matrix.`**`todense`**`(`*order=None*, *out=None*`)`

Return a dense matrix representation of this matrix.

Parameters

order : {'C', 'F'}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional

If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns

arr : numpy.matrix, 2-dimensional

A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

`bsr_matrix.`**`todia`**`(`*copy=False*`)`

Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

`bsr_matrix.`**`todok`**`(`*copy=False*`)`

Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

`bsr_matrix.tolil(copy=False)`
Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`bsr_matrix.transpose(axes=None, copy=False)`
Reverses the dimensions of the sparse matrix.

Parameters
axes : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.
copy : bool, optional
Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns
p : *self* with the dimensions reversed.

See also:

np.matrix.transpose
NumPy's implementation of 'transpose' for matrices

`bsr_matrix.trunc()`
Element-wise trunc.

See numpy.trunc for more information.

class `scipy.sparse.coo_matrix(arg1, shape=None, dtype=None, copy=False)`
A sparse matrix in COOrdinate format.

Also known as the 'ijv' or 'triplet' format.
This can be instantiated in several ways:

coo_matrix(D)
with a dense matrix D
coo_matrix(S) with another sparse matrix S (equivalent to S.tocoo())
coo_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.
coo_matrix((data, (i, j)), [shape=(M, N)])

to construct from three arrays:

1. data[:] the entries of the matrix, in any order
2. i[:] the row indices of the matrix entries
3. j[:] the column indices of the matrix entries

Where `A[i[k], j[k]] = data[k]`. When shape is not specified, it is inferred from the index arrays

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.
Advantages of the COO format

- facilitates fast conversion among sparse formats
- permits duplicate entries (see example)
- very fast conversion to and from CSR/CSC formats

Disadvantages of the COO format

- ***does not directly support:***
  - arithmetic operations
  - slicing

Intended Usage

- COO is a fast format for constructing sparse matrices
- Once a matrix has been constructed, convert to CSR or CSC format for fast arithmetic and matrix vector operations
- By default when converting to CSR or CSC format, duplicate (i,j) entries will be summed together. This facilitates efficient construction of finite element matrices and the like. (see example)

Examples

```
>>> from scipy.sparse import coo_matrix
>>> coo_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

```
>>> row  = np.array([0, 3, 1, 0])
>>> col  = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> coo_matrix((data, (row, col)), shape=(4, 4)).toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

```
>>> # example with duplicates
>>> row  = np.array([0, 0, 1, 3, 1, 0, 0])
>>> col  = np.array([0, 2, 1, 3, 1, 0, 0])
>>> data = np.array([1, 1, 1, 1, 1, 1, 1])
>>> coo_matrix((data, (row, col)), shape=(4, 4)).toarray()
array([[3, 0, 1, 0],
       [0, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 1]])
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.

`coo_matrix.`**`shape`**
Get shape of a matrix.

`coo_matrix.`**`nnz`**
Number of stored values, including explicit zeros.

See also:

`count_nonzero`

Number of non-zero entries

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	COO format data array of the matrix
row	COO format row index array of the matrix
col	COO format column index array of the matrix

Methods

arcsin()	Element-wise arcsin.
arcsinh()	Element-wise arcsinh.
arctan()	Element-wise arctan.
arctanh()	Element-wise arctanh.
argmax([axis, out])	Return indices of minimum elements along an axis.
argmin([axis, out])	Return indices of minimum elements along an axis.
asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
ceil()	Element-wise ceil.
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
deg2rad()	Element-wise deg2rad.
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
eliminate_zeros()	Remove zero entries from the matrix
expm1()	Element-wise expm1.
floor()	Element-wise floor.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).
log1p()	Element-wise log1p.
max([axis, out])	Return the maximum of the matrix or maximum along an axis.
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
min([axis, out])	Return the minimum of the matrix or maximum along an axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix
nonzero()	nonzero indices
power(n[, dtype])	This function performs element-wise power.

Continued on next page

Table 5.152 – continued from previous page

rad2deg()	Element-wise rad2deg.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
rint()	Element-wise rint.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sign()	Element-wise sign.
sin()	Element-wise sin.
sinh()	Element-wise sinh.
sqrt()	Element-wise sqrt.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
sum_duplicates()	Eliminate duplicate matrix entries by adding them together
tan()	Element-wise tan.
tanh()	Element-wise tanh.
toarray([order, out])	See the docstring for *spmatrix.toarray*.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format
tocsr([copy])	Convert this matrix to Compressed Sparse Row format
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
trunc()	Element-wise trunc.

coo_matrix.**arcsin**()

Element-wise arcsin.

See numpy.arcsin for more information.

coo_matrix.**arcsinh**()

Element-wise arcsinh.

See numpy.arcsinh for more information.

coo_matrix.**arctan**()

Element-wise arctan.

See numpy.arctan for more information.

coo_matrix.**arctanh**()

Element-wise arctanh.

See numpy.arctanh for more information.

coo_matrix.**argmax**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several maximum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int
Indices of maximum elements. If matrix, its size along *axis* is 1.

coo_matrix.**argmin**(*axis=None*, *out=None*)
Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several minimum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional
Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int
Indices of minimum elements. If matrix, its size along *axis* is 1.

coo_matrix.**asformat**(*format*)
Return this matrix in a given sparse format

Parameters **format** : {string, None}
desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

coo_matrix.**asfptype**()
Upcast matrix to a floating point format (if necessary)

coo_matrix.**astype**(*t*)
Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype
Typecode or data-type to which to cast the data.

coo_matrix.**ceil**()
Element-wise ceil.

See numpy.ceil for more information.

coo_matrix.**conj**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

coo_matrix.**conjugate**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`coo_matrix.copy()`
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

`coo_matrix.count_nonzero()`
Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

`coo_matrix.deg2rad()`
Element-wise deg2rad.

See numpy.deg2rad for more information.

`coo_matrix.diagonal()`
Returns the main diagonal of the matrix

`coo_matrix.dot(other)`
Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

`coo_matrix.eliminate_zeros()`
Remove zero entries from the matrix

This is an *in place* operation

`coo_matrix.expm1()`
Element-wise expm1.

See numpy.expm1 for more information.

`coo_matrix.floor()`
Element-wise floor.

See numpy.floor for more information.

`coo_matrix.getH()`
Return the Hermitian transpose of this matrix.

See also:

np.matrix.getH
NumPy's implementation of `getH` for matrices

`coo_matrix.get_shape()`
Get shape of a matrix.

`coo_matrix.getcol(j)`
Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

`coo_matrix.getformat()`
Format of a matrix representation as a string.

coo_matrix.**getmaxprint**()
Maximum number of elements to display when printed.

coo_matrix.**getnnz**(*axis=None*)
Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1
Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero
Number of non-zero entries

coo_matrix.**getrow**(*i*)
Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).

coo_matrix.**log1p**()
Element-wise log1p.

See numpy.log1p for more information.

coo_matrix.**max**(*axis=None*, *out=None*)
Return the maximum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amax** : coo_matrix or scalar
Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

min The minimum value of a sparse matrix along a given axis.

np.matrix.max
NumPy's implementation of 'max' for matrices

coo_matrix.**maximum**(*other*)
Element-wise maximum between this and another matrix.

coo_matrix.**mean**(*axis=None*, *dtype=None*, *out=None*)
Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional

Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean NumPy's implementation of 'mean' for matrices

`coo_matrix.`**`min`**`(`*axis=None*, *out=None*`)`

Return the minimum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional

Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amin** : coo_matrix or scalar

Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

`max` The maximum value of a sparse matrix along a given axis.

np.matrix.min NumPy's implementation of 'min' for matrices

`coo_matrix.`**`minimum`**`(`*other*`)`

Element-wise minimum between this and another matrix.

`coo_matrix.`**`multiply`**`(`*other*`)`

Point-wise multiplication by another matrix

`coo_matrix.`**`nonzero`**`()`

nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`coo_matrix.`**`power`**`(`*n*, *dtype=None*`)`

This function performs element-wise power.

Parameters **n** : n is a scalar

dtype : If dtype is not specified, the current dtype will be preserved.

`coo_matrix.`**`rad2deg`**`()`

Element-wise rad2deg.

See numpy.rad2deg for more information.

`coo_matrix.`**`reshape`**`(`*shape*, *order='C'*`)`

Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.
order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

np.matrix.reshape
NumPy's implementation of 'reshape' for matrices

coo_matrix.**rint**()
Element-wise rint.

See numpy.rint for more information.

coo_matrix.**set_shape**(*shape*)
See `reshape`.

coo_matrix.**setdiag**(*values*, *k=0*)
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.
k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

coo_matrix.**sign**()
Element-wise sign.

See numpy.sign for more information.

coo_matrix.**sin**()
Element-wise sin.

See numpy.sin for more information.

coo_matrix.**sinh**()
Element-wise sinh.

See numpy.sinh for more information.

coo_matrix.**sqrt**()
Element-wise sqrt.

See numpy.sqrt for more information.

coo_matrix.**sum**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).
dtype : dtype, optional

The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

coo_matrix.**sum_duplicates**()
Eliminate duplicate matrix entries by adding them together

This is an *in place* operation

coo_matrix.**tan**()
Element-wise tan.

See numpy.tan for more information.

coo_matrix.**tanh**()
Element-wise tanh.

See numpy.tanh for more information.

coo_matrix.**toarray**(*order=None*, *out=None*)
See the docstring for *spmatrix.toarray*.

coo_matrix.**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

coo_matrix.**tocoo**(*copy=False*)
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

coo_matrix.**tocsc**(*copy=False*)
Convert this matrix to Compressed Sparse Column format

Duplicate entries will be summed together.

Examples

```
>>> from numpy import array
>>> from scipy.sparse import coo_matrix
>>> row  = array([0, 0, 1, 3, 1, 0, 0])
>>> col  = array([0, 2, 1, 3, 1, 0, 0])
>>> data = array([1, 1, 1, 1, 1, 1, 1])
>>> A = coo_matrix((data, (row, col)), shape=(4, 4)).tocsc()
>>> A.toarray()
array([[3, 0, 1, 0],
```

```
       [0, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 1]])
```

coo_matrix.**tocsr**(*copy=False*)

Convert this matrix to Compressed Sparse Row format

Duplicate entries will be summed together.

Examples

```
>>> from numpy import array
>>> from scipy.sparse import coo_matrix
>>> row  = array([0, 0, 1, 3, 1, 0, 0])
>>> col  = array([0, 2, 1, 3, 1, 0, 0])
>>> data = array([1, 1, 1, 1, 1, 1, 1])
>>> A = coo_matrix((data, (row, col)), shape=(4, 4)).tocsr()
>>> A.toarray()
array([[3, 0, 1, 0],
       [0, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 1]])
```

coo_matrix.**todense**(*order=None*, *out=None*)

Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional

If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional

A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

coo_matrix.**todia**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

coo_matrix.**todok**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

coo_matrix.**tolil**(*copy=False*)

Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

coo_matrix.**transpose**(*axes=None*, *copy=False*)

Reverses the dimensions of the sparse matrix.

Parameters **axes** : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional

Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p** : *self* with the dimensions reversed.

See also:

np.matrix.transpose
NumPy's implementation of 'transpose' for matrices

`coo_matrix.trunc()`
Element-wise trunc.

See numpy.trunc for more information.

class `scipy.sparse.csc_matrix`(*arg1*, *shape=None*, *dtype=None*, *copy=False*)
Compressed Sparse Column matrix

This can be instantiated in several ways:

csc_matrix(D) with a dense matrix or rank-2 ndarray D

csc_matrix(S) with another sparse matrix S (equivalent to S.tocsc())

csc_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

csc_matrix((data, (row_ind, col_ind)), [shape=(M, N)])
where `data`, `row_ind` and `col_ind` satisfy the relationship `a[row_ind[k], col_ind[k]] = data[k]`.

csc_matrix((data, indices, indptr), [shape=(M, N)])
is the standard CSC representation where the row indices for column i are stored in `indices[indptr[i]:indptr[i+1]]` and their corresponding values are stored in `data[indptr[i]:indptr[i+1]]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Advantages of the CSC format

- efficient arithmetic operations CSC + CSC, CSC * CSC, etc.
- efficient column slicing
- fast matrix vector products (CSR, BSR may be faster)

Disadvantages of the CSC format

- slow row slicing operations (consider CSR)
- changes to the sparsity structure are expensive (consider LIL or DOK)

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csc_matrix
>>> csc_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

```
>>> row = np.array([0, 2, 2, 0, 1, 2])
>>> col = np.array([0, 0, 1, 2, 2, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csc_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

```
>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csc_matrix((data, indices, indptr), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.
has_sorted_indices	Determine whether the matrix has sorted indices

csc_matrix.**shape**
Get shape of a matrix.

csc_matrix.**nnz**
Number of stored values, including explicit zeros.

See also:

count_nonzero
Number of non-zero entries

csc_matrix.**has_sorted_indices**
Determine whether the matrix has sorted indices
Returns

- True: if the indices of the matrix are in sorted order
- False: otherwise

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	Data array of the matrix
indices	CSC format index array
indptr	CSC format index pointer array

Methods

arcsin()	Element-wise arcsin.
arcsinh()	Element-wise arcsinh.
arctan()	Element-wise arctan.
arctanh()	Element-wise arctanh.
argmax([axis, out])	Return indices of minimum elements along an axis.
argmin([axis, out])	Return indices of minimum elements along an axis.
asformat(format)	Return this matrix in a given sparse format

Continued on next page

Table 5.154 – continued from previous page

asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
ceil()	Element-wise ceil.
check_format([full_check])	check whether the matrix format is valid
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
deg2rad()	Element-wise deg2rad.
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
eliminate_zeros()	Remove zero entries from the matrix
expm1()	Element-wise expm1.
floor()	Element-wise floor.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(i)	Returns a copy of column i of the matrix, as a (m x 1) CSC matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) CSR matrix (row vector).
log1p()	Element-wise log1p.
max([axis, out])	Return the maximum of the matrix or maximum along an axis.
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
min([axis, out])	Return the minimum of the matrix or maximum along an axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix, vector, or scalar.
nonzero()	nonzero indices
power(n[, dtype])	This function performs element-wise power.
prune()	Remove empty space after all non-zero elements.
rad2deg()	Element-wise rad2deg.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
rint()	Element-wise rint.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sign()	Element-wise sign.
sin()	Element-wise sin.
sinh()	Element-wise sinh.
sort_indices()	Sort the indices of this matrix *in place*
sorted_indices()	Return a copy of this matrix with sorted indices
sqrt()	Element-wise sqrt.

Continued on next page

Table 5.154 – continued from previous page

sum([axis, dtype, out])	Sum the matrix elements over a given axis.
sum_duplicates()	Eliminate duplicate matrix entries by adding them together
tan()	Element-wise tan.
tanh()	Element-wise tanh.
toarray([order, out])	See the docstring for *spmatrix.toarray*.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.
tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
trunc()	Element-wise trunc.

csc_matrix.**arcsin**()

Element-wise arcsin.

See numpy.arcsin for more information.

csc_matrix.**arcsinh**()

Element-wise arcsinh.

See numpy.arcsinh for more information.

csc_matrix.**arctan**()

Element-wise arctan.

See numpy.arctan for more information.

csc_matrix.**arctanh**()

Element-wise arctanh.

See numpy.arctanh for more information.

csc_matrix.**argmax**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several maximum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int

Indices of maximum elements. If matrix, its size along *axis* is 1.

csc_matrix.**argmin**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several minimum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional
Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int
Indices of minimum elements. If matrix, its size along *axis* is 1.

csc_matrix.**asformat**(*format*)
Return this matrix in a given sparse format

Parameters **format** : {string, None}
desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

csc_matrix.**asfptype**()
Upcast matrix to a floating point format (if necessary)

csc_matrix.**astype**(*t*)
Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype
Typecode or data-type to which to cast the data.

csc_matrix.**ceil**()
Element-wise ceil.

See numpy.ceil for more information.

csc_matrix.**check_format**(*full_check=True*)
check whether the matrix format is valid

Parameters **full_check** : bool, optional
If *True*, rigorous check, O(N) operations. Otherwise basic check, O(1) operations (default True).

csc_matrix.**conj**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

csc_matrix.**conjugate**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

csc_matrix.**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

csc_matrix.**count_nonzero**()
Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

`csc_matrix.`**`deg2rad`**`()`

Element-wise deg2rad.

See numpy.deg2rad for more information.

`csc_matrix.`**`diagonal`**`()`

Returns the main diagonal of the matrix

`csc_matrix.`**`dot`**`(`*other*`)`

Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

`csc_matrix.`**`eliminate_zeros`**`()`

Remove zero entries from the matrix

This is an *in place* operation

`csc_matrix.`**`expm1`**`()`

Element-wise expm1.

See numpy.expm1 for more information.

`csc_matrix.`**`floor`**`()`

Element-wise floor.

See numpy.floor for more information.

`csc_matrix.`**`getH`**`()`

Return the Hermitian transpose of this matrix.

See also:

np.matrix.getH

NumPy's implementation of *getH* for matrices

`csc_matrix.`**`get_shape`**`()`

Get shape of a matrix.

`csc_matrix.`**`getcol`**`(`*i*`)`

Returns a copy of column i of the matrix, as a (m x 1) CSC matrix (column vector).

`csc_matrix.`**`getformat`**`()`

Format of a matrix representation as a string.

`csc_matrix.`**`getmaxprint`**`()`

Maximum number of elements to display when printed.

`csc_matrix.`**`getnnz`**`(`*axis=None*`)`

Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1

Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero
Number of non-zero entries

`csc_matrix.`**getrow**(*i*)

Returns a copy of row i of the matrix, as a (1 x n) CSR matrix (row vector).

`csc_matrix.`**log1p**()

Element-wise log1p.

See numpy.log1p for more information.

`csc_matrix.`**max**(*axis=None*, *out=None*)

Return the maximum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amax** : coo_matrix or scalar
Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

min The minimum value of a sparse matrix along a given axis.

np.matrix.max
NumPy's implementation of 'max' for matrices

`csc_matrix.`**maximum**(*other*)

Element-wise maximum between this and another matrix.

`csc_matrix.`**mean**(*axis=None*, *dtype=None*, *out=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean
NumPy's implementation of 'mean' for matrices

csc_matrix.**min**(*axis=None*, *out=None*)
Return the minimum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amin** : coo_matrix or scalar
Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

max The maximum value of a sparse matrix along a given axis.

np.matrix.min
NumPy's implementation of 'min' for matrices

csc_matrix.**minimum**(*other*)
Element-wise minimum between this and another matrix.

csc_matrix.**multiply**(*other*)
Point-wise multiplication by another matrix, vector, or scalar.

csc_matrix.**nonzero**()
nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

csc_matrix.**power**(*n*, *dtype=None*)
This function performs element-wise power.

Parameters **n** : n is a scalar

dtype : If dtype is not specified, the current dtype will be preserved.

csc_matrix.**prune**()
Remove empty space after all non-zero elements.

csc_matrix.**rad2deg**()
Element-wise rad2deg.

See numpy.rad2deg for more information.

csc_matrix.**reshape**(*shape*, *order='C'*)
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.

order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

np.matrix.reshape
NumPy's implementation of 'reshape' for matrices

csc_matrix.**rint**()
Element-wise rint.

See numpy.rint for more information.

csc_matrix.**set_shape**(*shape*)
See `reshape`.

csc_matrix.**setdiag**(*values*, *k=0*)
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.

k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

csc_matrix.**sign**()
Element-wise sign.

See numpy.sign for more information.

csc_matrix.**sin**()
Element-wise sin.

See numpy.sin for more information.

csc_matrix.**sinh**()
Element-wise sinh.

See numpy.sinh for more information.

csc_matrix.**sort_indices**()
Sort the indices of this matrix *in place*

csc_matrix.**sorted_indices**()
Return a copy of this matrix with sorted indices

csc_matrix.**sqrt**()
Element-wise sqrt.

See numpy.sqrt for more information.

csc_matrix.**sum**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).

dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

`csc_matrix.`**`sum_duplicates`**`()`
Eliminate duplicate matrix entries by adding them together

The is an *in place* operation

`csc_matrix.`**`tan`**`()`
Element-wise tan.

See numpy.tan for more information.

`csc_matrix.`**`tanh`**`()`
Element-wise tanh.

See numpy.tanh for more information.

`csc_matrix.`**`toarray`**`(`*order=None*, *out=None*`)`
See the docstring for *`spmatrix.toarray`*.

`csc_matrix.`**`tobsr`**`(`*blocksize=None*, *copy=False*`)`
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

`csc_matrix.`**`tocoo`**`(`*copy=True*`)`
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`csc_matrix.`**`tocsc`**`(`*copy=False*`)`
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

`csc_matrix.`**`tocsr`**`(`*copy=False*`)`
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

`csc_matrix.`**`todense`**`(`*order=None*, *out=None*`)`
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional
If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns
arr : numpy.matrix, 2-dimensional
A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

`csc_matrix.`**`todia`**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

`csc_matrix.`**`todok`**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

`csc_matrix.`**`tolil`**(*copy=False*)
Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`csc_matrix.`**`transpose`**(*axes=None*, *copy=False*)
Reverses the dimensions of the sparse matrix.

Parameters
axes : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional
Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns
p : *self* with the dimensions reversed.

See also:

`np.matrix.transpose`
NumPy's implementation of 'transpose' for matrices

`csc_matrix.`**`trunc`**()
Element-wise trunc.

See numpy.trunc for more information.

class `scipy.sparse.`**`csr_matrix`**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)
Compressed Sparse Row matrix
This can be instantiated in several ways:

csr_matrix(D) with a dense matrix or rank-2 ndarray D
csr_matrix(S) with another sparse matrix S (equivalent to S.tocsr())

csr_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

csr_matrix((data, (row_ind, col_ind)), [shape=(M, N)])
where `data`, `row_ind` and `col_ind` satisfy the relationship `a[row_ind[k], col_ind[k]] = data[k]`.

csr_matrix((data, indices, indptr), [shape=(M, N)])
is the standard CSR representation where the column indices for row i are stored in `indices[indptr[i]:indptr[i+1]]` and their corresponding values are stored in `data[indptr[i]:indptr[i+1]]`. If the shape parameter is not supplied, the matrix dimensions are inferred from the index arrays.

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Advantages of the CSR format

- efficient arithmetic operations CSR + CSR, CSR * CSR, etc.
- efficient row slicing
- fast matrix vector products

Disadvantages of the CSR format

- slow column slicing operations (consider CSC)
- changes to the sparsity structure are expensive (consider LIL or DOK)

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> csr_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csr_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

```
>>> indptr = np.array([0, 2, 3, 6])
>>> indices = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csr_matrix((data, indices, indptr), shape=(3, 3)).toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

As an example of how to construct a CSR matrix incrementally, the following snippet builds a term-document matrix from texts:

```
>>> docs = [["hello", "world", "hello"], ["goodbye", "cruel", "world"]]
>>> indptr = [0]
```

```
>>> indices = []
>>> data = []
>>> vocabulary = {}
>>> for d in docs:
...     for term in d:
...         index = vocabulary.setdefault(term, len(vocabulary))
...         indices.append(index)
...         data.append(1)
...     indptr.append(len(indices))
...
>>> csr_matrix((data, indices, indptr), dtype=int).toarray()
array([[2, 1, 0, 0],
       [0, 1, 1, 1]])
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.
has_sorted_indices	Determine whether the matrix has sorted indices

csr_matrix.**shape**
Get shape of a matrix.

csr_matrix.**nnz**
Number of stored values, including explicit zeros.

See also:

count_nonzero
Number of non-zero entries

csr_matrix.**has_sorted_indices**
Determine whether the matrix has sorted indices
Returns

- True: if the indices of the matrix are in sorted order
- False: otherwise

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	CSR format data array of the matrix
indices	CSR format index array of the matrix
indptr	CSR format index pointer array of the matrix

Methods

arcsin()	Element-wise arcsin.
arcsinh()	Element-wise arcsinh.
arctan()	Element-wise arctan.
arctanh()	Element-wise arctanh.
argmax([axis, out])	Return indices of minimum elements along an axis.
argmin([axis, out])	Return indices of minimum elements along an axis.
asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.

Continued on next page

Table 5.156 – continued from previous page

ceil()	Element-wise ceil.
check_format([full_check])	check whether the matrix format is valid
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
deg2rad()	Element-wise deg2rad.
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
eliminate_zeros()	Remove zero entries from the matrix
expm1()	Element-wise expm1.
floor()	Element-wise floor.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(i)	Returns a copy of column i of the matrix, as a (m x 1) CSR matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) CSR matrix (row vector).
log1p()	Element-wise log1p.
max([axis, out])	Return the maximum of the matrix or maximum along an axis.
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
min([axis, out])	Return the minimum of the matrix or maximum along an axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix, vector, or scalar.
nonzero()	nonzero indices
power(n[, dtype])	This function performs element-wise power.
prune()	Remove empty space after all non-zero elements.
rad2deg()	Element-wise rad2deg.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
rint()	Element-wise rint.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sign()	Element-wise sign.
sin()	Element-wise sin.
sinh()	Element-wise sinh.
sort_indices()	Sort the indices of this matrix *in place*
sorted_indices()	Return a copy of this matrix with sorted indices
sqrt()	Element-wise sqrt.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.

Continued on next page

Table 5.156 – continued from previous page

sum_duplicates()	Eliminate duplicate matrix entries by adding them together
tan()	Element-wise tan.
tanh()	Element-wise tanh.
toarray([order, out])	See the docstring for *spmatrix.toarray*.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.
tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
trunc()	Element-wise trunc.

csr_matrix.**arcsin**()

Element-wise arcsin.

See numpy.arcsin for more information.

csr_matrix.**arcsinh**()

Element-wise arcsinh.

See numpy.arcsinh for more information.

csr_matrix.**arctan**()

Element-wise arctan.

See numpy.arctan for more information.

csr_matrix.**arctanh**()

Element-wise arctanh.

See numpy.arctanh for more information.

csr_matrix.**argmax**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several maximum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmax is computed. If None (default), index of the maximum element in the flatten data is returned.

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int

Indices of maximum elements. If matrix, its size along *axis* is 1.

csr_matrix.**argmin**(*axis=None*, *out=None*)

Return indices of minimum elements along an axis.

Implicit zero elements are also taken into account. If there are several minimum values, the index of the first occurrence is returned.

Parameters **axis** : {-2, -1, 0, 1, None}, optional

Axis along which the argmin is computed. If None (default), index of the minimum element in the flatten data is returned.

out : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **ind** : np.matrix or int

Indices of minimum elements. If matrix, its size along *axis* is 1.

`csr_matrix.`**`asformat`**`(`*format*`)`

Return this matrix in a given sparse format

Parameters **format** : {string, None}

desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

`csr_matrix.`**`asfptype`**`()`

Upcast matrix to a floating point format (if necessary)

`csr_matrix.`**`astype`**`(`*t*`)`

Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype

Typecode or data-type to which to cast the data.

`csr_matrix.`**`ceil`**`()`

Element-wise ceil.

See numpy.ceil for more information.

`csr_matrix.`**`check_format`**`(`*full_check=True*`)`

check whether the matrix format is valid

Parameters **full_check** : bool, optional

If *True*, rigorous check, O(N) operations. Otherwise basic check, O(1) operations (default True).

`csr_matrix.`**`conj`**`()`

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`csr_matrix.`**`conjugate`**`()`

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`csr_matrix.`**`copy`**`()`

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

`csr_matrix.`**`count_nonzero`**`()`

Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

csr_matrix.**deg2rad**()

Element-wise deg2rad.

See numpy.deg2rad for more information.

csr_matrix.**diagonal**()

Returns the main diagonal of the matrix

csr_matrix.**dot**(*other*)

Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

csr_matrix.**eliminate_zeros**()

Remove zero entries from the matrix

This is an *in place* operation

csr_matrix.**expm1**()

Element-wise expm1.

See numpy.expm1 for more information.

csr_matrix.**floor**()

Element-wise floor.

See numpy.floor for more information.

csr_matrix.**getH**()

Return the Hermitian transpose of this matrix.

See also:

np.matrix.getH NumPy's implementation of *getH* for matrices

csr_matrix.**get_shape**()

Get shape of a matrix.

csr_matrix.**getcol**(*i*)

Returns a copy of column i of the matrix, as a (m x 1) CSR matrix (column vector).

csr_matrix.**getformat**()

Format of a matrix representation as a string.

csr_matrix.**getmaxprint**()

Maximum number of elements to display when printed.

csr_matrix.**getnnz**(*axis=None*)

Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1

Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero
Number of non-zero entries

csr_matrix.**getrow**(*i*)
Returns a copy of row i of the matrix, as a (1 x n) CSR matrix (row vector).

csr_matrix.**log1p**()
Element-wise log1p.

See numpy.log1p for more information.

csr_matrix.**max**(*axis=None*, *out=None*)
Return the maximum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the maximum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amax** : coo_matrix or scalar
Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

min The minimum value of a sparse matrix along a given axis.

np.matrix.max
NumPy's implementation of 'max' for matrices

csr_matrix.**maximum**(*other*)
Element-wise maximum between this and another matrix.

csr_matrix.**mean**(*axis=None*, *dtype=None*, *out=None*)
Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean
NumPy's implementation of 'mean' for matrices

`csr_matrix.`**`min`**(*axis=None*, *out=None*)
Return the minimum of the matrix or maximum along an axis. This takes all elements into account, not just the non-zero ones.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the minimum over all the matrix elements, returning a scalar (i.e. *axis* = *None*).

out : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **amin** : coo_matrix or scalar
Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is a sparse.coo_matrix of dimension `a.ndim - 1`.

See also:

`max` The maximum value of a sparse matrix along a given axis.

np.matrix.min
NumPy's implementation of 'min' for matrices

`csr_matrix.`**`minimum`**(*other*)
Element-wise minimum between this and another matrix.

`csr_matrix.`**`multiply`**(*other*)
Point-wise multiplication by another matrix, vector, or scalar.

`csr_matrix.`**`nonzero`**()
nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`csr_matrix.`**`power`**(*n*, *dtype=None*)
This function performs element-wise power.

Parameters **n** : n is a scalar
dtype : If dtype is not specified, the current dtype will be preserved.

`csr_matrix.`**`prune`**()
Remove empty space after all non-zero elements.

`csr_matrix.`**`rad2deg`**()
Element-wise rad2deg.

See numpy.rad2deg for more information.

`csr_matrix.`**`reshape`**(*shape*, *order='C'*)
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.

order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

np.matrix.reshape
NumPy's implementation of 'reshape' for matrices

csr_matrix.**rint**()
Element-wise rint.

See numpy.rint for more information.

csr_matrix.**set_shape**(*shape*)
See `reshape`.

csr_matrix.**setdiag**(*values*, *k=0*)
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.

k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

csr_matrix.**sign**()
Element-wise sign.

See numpy.sign for more information.

csr_matrix.**sin**()
Element-wise sin.

See numpy.sin for more information.

csr_matrix.**sinh**()
Element-wise sinh.

See numpy.sinh for more information.

csr_matrix.**sort_indices**()
Sort the indices of this matrix *in place*

csr_matrix.**sorted_indices**()
Return a copy of this matrix with sorted indices

csr_matrix.**sqrt**()
Element-wise sqrt.

See numpy.sqrt for more information.

csr_matrix.**sum**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).

dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

csr_matrix.**sum_duplicates**()
Eliminate duplicate matrix entries by adding them together

The is an *in place* operation

csr_matrix.**tan**()
Element-wise tan.

See numpy.tan for more information.

csr_matrix.**tanh**()
Element-wise tanh.

See numpy.tanh for more information.

csr_matrix.**toarray**(*order=None*, *out=None*)
See the docstring for *spmatrix.toarray*.

csr_matrix.**tobsr**(*blocksize=None*, *copy=True*)
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

csr_matrix.**tocoo**(*copy=True*)
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

csr_matrix.**tocsc**(*copy=False*)
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

csr_matrix.**tocsr**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

csr_matrix.**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional

Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional
If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional
A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

`csr_matrix.`**`todia`**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

`csr_matrix.`**`todok`**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

`csr_matrix.`**`tolil`**(*copy=False*)
Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`csr_matrix.`**`transpose`**(*axes=None*, *copy=False*)
Reverses the dimensions of the sparse matrix.

Parameters **axes** : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional
Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p** : *self* with the dimensions reversed.

See also:

`np.matrix.transpose`
NumPy's implementation of 'transpose' for matrices

`csr_matrix.`**`trunc`**()
Element-wise trunc.

See numpy.trunc for more information.

class `scipy.sparse.`**`dia_matrix`**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)
Sparse matrix with DIAgonal storage
This can be instantiated in several ways:

dia_matrix(D) with a dense matrix
dia_matrix(S) with another sparse matrix S (equivalent to S.todia())

dia_matrix((M, N), [dtype])
to construct an empty matrix with shape (M, N), dtype is optional, defaulting to dtype='d'.

dia_matrix((data, offsets), shape=(M, N))
where the `data[k,:]` stores the diagonal entries for diagonal `offsets[k]` (See example below)

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Examples

```
>>> import numpy as np
>>> from scipy.sparse import dia_matrix
>>> dia_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

```
>>> data = np.array([[1, 2, 3, 4]]).repeat(3, axis=0)
>>> offsets = np.array([0, -1, 2])
>>> dia_matrix((data, offsets), shape=(4, 4)).toarray()
array([[1, 0, 3, 0],
       [1, 2, 0, 4],
       [0, 2, 3, 0],
       [0, 0, 3, 4]])
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.

`dia_matrix.`**`shape`**
Get shape of a matrix.

`dia_matrix.`**`nnz`**
Number of stored values, including explicit zeros.

See also:

`count_nonzero`
Number of non-zero entries

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	DIA format data array of the matrix
offsets	DIA format offset array of the matrix

Methods

arcsin()	Element-wise arcsin.
arcsinh()	Element-wise arcsinh.
arctan()	Element-wise arctan.

Continued on next page

Table 5.158 – continued from previous page

arctanh()	Element-wise arctanh.
asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
ceil()	Element-wise ceil.
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
deg2rad()	Element-wise deg2rad.
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
expm1()	Element-wise expm1.
floor()	Element-wise floor.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).
log1p()	Element-wise log1p.
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix
nonzero()	nonzero indices
power(n[, dtype])	This function performs element-wise power.
rad2deg()	Element-wise rad2deg.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
rint()	Element-wise rint.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sign()	Element-wise sign.
sin()	Element-wise sin.
sinh()	Element-wise sinh.
sqrt()	Element-wise sqrt.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
tan()	Element-wise tan.
tanh()	Element-wise tanh.
toarray([order, out])	Return a dense ndarray representation of this matrix.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.

Continued on next page

Table 5.158 – continued from previous page

tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
trunc()	Element-wise trunc.

dia_matrix.**arcsin**()

Element-wise arcsin.

See numpy.arcsin for more information.

dia_matrix.**arcsinh**()

Element-wise arcsinh.

See numpy.arcsinh for more information.

dia_matrix.**arctan**()

Element-wise arctan.

See numpy.arctan for more information.

dia_matrix.**arctanh**()

Element-wise arctanh.

See numpy.arctanh for more information.

dia_matrix.**asformat**(*format*)

Return this matrix in a given sparse format

Parameters **format** : {string, None}

desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

dia_matrix.**asfptype**()

Upcast matrix to a floating point format (if necessary)

dia_matrix.**astype**(*t*)

Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype

Typecode or data-type to which to cast the data.

dia_matrix.**ceil**()

Element-wise ceil.

See numpy.ceil for more information.

dia_matrix.**conj**()

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`dia_matrix.`**`conjugate`**`()`

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`dia_matrix.`**`copy`**`()`

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

`dia_matrix.`**`count_nonzero`**`()`

Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

`dia_matrix.`**`deg2rad`**`()`

Element-wise deg2rad.

See numpy.deg2rad for more information.

`dia_matrix.`**`diagonal`**`()`

Returns the main diagonal of the matrix

`dia_matrix.`**`dot`**`(`*other*`)`

Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

`dia_matrix.`**`expm1`**`()`

Element-wise expm1.

See numpy.expm1 for more information.

`dia_matrix.`**`floor`**`()`

Element-wise floor.

See numpy.floor for more information.

`dia_matrix.`**`getH`**`()`

Return the Hermitian transpose of this matrix.

See also:

`np.matrix.getH`
NumPy's implementation of *`getH`* for matrices

`dia_matrix.`**`get_shape`**`()`

Get shape of a matrix.

`dia_matrix.`**`getcol`**`(`*j*`)`

Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

`dia_matrix.`**`getformat`**`()`

Format of a matrix representation as a string.

dia_matrix.**getmaxprint**()

Maximum number of elements to display when printed.

dia_matrix.**getnnz**(*axis=None*)

Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1

Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero

Number of non-zero entries

dia_matrix.**getrow**(*i*)

Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).

dia_matrix.**log1p**()

Element-wise log1p.

See numpy.log1p for more information.

dia_matrix.**maximum**(*other*)

Element-wise maximum between this and another matrix.

dia_matrix.**mean**(*axis=None*, *dtype=None*, *out=None*)

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional

Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional

Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional

Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean

NumPy's implementation of 'mean' for matrices

dia_matrix.**minimum**(*other*)

Element-wise minimum between this and another matrix.

dia_matrix.**multiply**(*other*)

Point-wise multiplication by another matrix

dia_matrix.**nonzero**()

nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`dia_matrix.`**`power`**(*n*, *dtype=None*)
This function performs element-wise power.

Parameters **n** : n is a scalar
dtype : If dtype is not specified, the current dtype will be preserved.

`dia_matrix.`**`rad2deg`**()
Element-wise rad2deg.

See numpy.rad2deg for more information.

`dia_matrix.`**`reshape`**(*shape*, *order='C'*)
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.
order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

`np.matrix.reshape`
NumPy's implementation of 'reshape' for matrices

`dia_matrix.`**`rint`**()
Element-wise rint.

See numpy.rint for more information.

`dia_matrix.`**`set_shape`**(*shape*)
See `reshape`.

`dia_matrix.`**`setdiag`**(*values*, *k=0*)
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.
k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

`dia_matrix.`**`sign`**()
Element-wise sign.

See numpy.sign for more information.

`dia_matrix.`**`sin`**()
Element-wise sin.

See numpy.sin for more information.

dia_matrix.**sinh**()
Element-wise sinh.

See numpy.sinh for more information.

dia_matrix.**sqrt**()
Element-wise sqrt.

See numpy.sqrt for more information.

dia_matrix.**sum**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters
axis : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).
dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.
out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns
sum_along_axis : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

dia_matrix.**tan**()
Element-wise tan.

See numpy.tan for more information.

dia_matrix.**tanh**()
Element-wise tanh.

See numpy.tanh for more information.

dia_matrix.**toarray**(*order=None*, *out=None*)
Return a dense ndarray representation of this matrix.

Parameters
order : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.
out : ndarray, 2-dimensional, optional
If specified, uses this array as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method. For most sparse types, *out* is required to be memory contiguous (either C or Fortran ordered).

Returns
arr : ndarray, 2-dimensional

An array with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed, the same object is returned after being modified in-place to contain the appropriate values.

dia_matrix.**tobsr**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

dia_matrix.**tocoo**(*copy=False*)
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

dia_matrix.**tocsc**(*copy=False*)
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

dia_matrix.**tocsr**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

dia_matrix.**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional
If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional
A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

dia_matrix.**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

dia_matrix.**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

dia_matrix.**tolil**(*copy=False*)
Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`dia_matrix.transpose(axes=None, copy=False)`
Reverses the dimensions of the sparse matrix.

Parameters
axes : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.
copy : bool, optional
Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns
p : *self* with the dimensions reversed.

See also:

np.matrix.transpose
NumPy's implementation of 'transpose' for matrices

`dia_matrix.trunc()`
Element-wise trunc.

See numpy.trunc for more information.

class `scipy.sparse.dok_matrix(arg1, shape=None, dtype=None, copy=False)`
Dictionary Of Keys based sparse matrix.

This is an efficient structure for constructing sparse matrices incrementally.
This can be instantiated in several ways:

dok_matrix(D)
with a dense matrix, D
dok_matrix(S) with a sparse matrix, S
dok_matrix((M,N), [dtype])
create the matrix with initial shape (M,N) dtype is optional, defaulting to dtype='d'

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Allows for efficient O(1) access of individual elements. Duplicates are not allowed. Can be efficiently converted to a coo_matrix once constructed.

Examples

```
>>> import numpy as np
>>> from scipy.sparse import dok_matrix
>>> S = dok_matrix((5, 5), dtype=np.float32)
>>> for i in range(5):
...     for j in range(5):
...         S[i, j] = i + j    # Update element
```

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.

`dok_matrix.shape`
Get shape of a matrix.

dok_matrix.**nnz**

Number of stored values, including explicit zeros.

See also:

count_nonzero

Number of non-zero entries

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)

Methods

asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
clear(() -> None. Remove all items from D.)	
conj()	Element-wise complex conjugation.
conjtransp()	Return the conjugate transpose
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
fromkeys(...)	v defaults to None.
get(key[, default])	This overrides the dict.get method, providing type checking but otherwise equivalent functionality.
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix as a (m x 1) DOK matrix.
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix as a (1 x n) DOK matrix.
has_key((k) -> True if D has a key k, else False)	
items(() -> list of D's (key, value) pairs, ...)	
iteritems(() -> an iterator over the (key, ...)	
iterkeys(() -> an iterator over the keys of D)	
itervalues(...)	
keys(() -> list of D's keys)	
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix
nonzero()	nonzero indices
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
power(n[, dtype])	Element-wise power.

Continued on next page

Table 5.160 – continued from previous page

reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
resize(shape)	Resize the matrix in-place to dimensions given by 'shape'.
set_shape(shape)	See *reshape*.
setdefault((k[,d]) -> D.get(k,d), ...)	
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
toarray([order, out])	Return a dense ndarray representation of this matrix.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.
tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.
update(([E, ...)	If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values(() -> list of D's values)	
viewitems(...)	
viewkeys(...)	
viewvalues(...)	

dok_matrix.**asformat**(*format*)

Return this matrix in a given sparse format

Parameters **format** : {string, None}

desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

dok_matrix.**asfptype**()

Upcast matrix to a floating point format (if necessary)

dok_matrix.**astype**(*t*)

Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype

Typecode or data-type to which to cast the data.

dok_matrix.**clear**() → None. Remove all items from D.

dok_matrix.**conj**()

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

dok_matrix.**conjtransp**()
Return the conjugate transpose

dok_matrix.**conjugate**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

dok_matrix.**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

dok_matrix.**count_nonzero**()
Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

dok_matrix.**diagonal**()
Returns the main diagonal of the matrix

dok_matrix.**dot**(*other*)
Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

dok_matrix.**fromkeys**(S[, v]) → New dict with keys from S and values equal to v.
v defaults to None.

dok_matrix.**get**(*key*, *default=0.0*)
This overrides the dict.get method, providing type checking but otherwise equivalent functionality.

dok_matrix.**getH**()
Return the Hermitian transpose of this matrix.

See also:

np.matrix.getH NumPy's implementation of *getH* for matrices

dok_matrix.**get_shape**()
Get shape of a matrix.

dok_matrix.**getcol**(*j*)
Returns a copy of column j of the matrix as a (m x 1) DOK matrix.

dok_matrix.**getformat**()
Format of a matrix representation as a string.

dok_matrix.**getmaxprint**()
Maximum number of elements to display when printed.

dok_matrix.**getnnz**(*axis=None*)
Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1
Select between the number of values across the whole matrix, in each column, or in each row.

See also:

count_nonzero
Number of non-zero entries

dok_matrix.**getrow**(*i*)
Returns a copy of row i of the matrix as a (1 x n) DOK matrix.

dok_matrix.**has_key**(*k*) → True if D has a key k, else False

dok_matrix.**items**() → list of D's (key, value) pairs, as 2-tuples

dok_matrix.**iteritems**() → an iterator over the (key, value) items of D

dok_matrix.**iterkeys**() → an iterator over the keys of D

dok_matrix.**itervalues**() → an iterator over the values of D

dok_matrix.**keys**() → list of D's keys

dok_matrix.**maximum**(*other*)
Element-wise maximum between this and another matrix.

dok_matrix.**mean**(*axis=None*, *dtype=None*, *out=None*)
Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean
NumPy's implementation of 'mean' for matrices

dok_matrix.**minimum**(*other*)
Element-wise minimum between this and another matrix.

dok_matrix.**multiply**(*other*)
Point-wise multiplication by another matrix

`dok_matrix.`**`nonzero`**`()`

nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`dok_matrix.`**`pop`**`(`*k*[, *d*]`)` → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

`dok_matrix.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a

2-tuple; but raise KeyError if D is empty.

`dok_matrix.`**`power`**`(`*n*, *dtype=None*`)`

Element-wise power.

`dok_matrix.`**`reshape`**`(`*shape*, *order='C'*`)`

Gives a new shape to a sparse matrix without changing its data.

Parameters

shape : length-2 tuple of ints

The new shape should be compatible with the original shape.

order : 'C', optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns

reshaped_matrix : *self* with the new dimensions of `shape`

See also:

`np.matrix.reshape`

NumPy's implementation of 'reshape' for matrices

`dok_matrix.`**`resize`**`(`*shape*`)`

Resize the matrix in-place to dimensions given by 'shape'.

Any non-zero elements that lie outside the new shape are removed.

`dok_matrix.`**`set_shape`**`(`*shape*`)`

See `reshape`.

`dok_matrix.`**`setdefault`**`(`*k*[, *d*]`)` → D.get(k,d), also set D[k]=d if k not in D

`dok_matrix.`**`setdiag`**`(`*values*, *k=0*`)`

Set diagonal or off-diagonal elements of the array.

Parameters

values : array_like

New values of the diagonal elements.

Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.

If a scalar value is given, all of the diagonal is set to it.

k : int, optional

Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

`dok_matrix.sum(axis=None, dtype=None, out=None)`
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).

dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

`dok_matrix.toarray(order=None, out=None)`
Return a dense ndarray representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional
If specified, uses this array as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method. For most sparse types, *out* is required to be memory contiguous (either C or Fortran ordered).

Returns **arr** : ndarray, 2-dimensional
An array with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed, the same object is returned after being modified in-place to contain the appropriate values.

`dok_matrix.tobsr(blocksize=None, copy=False)`
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

`dok_matrix.tocoo(copy=False)`
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`dok_matrix.tocsc(copy=False)`
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

dok_matrix.**tocsr**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

dok_matrix.**todense**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.

out : ndarray, 2-dimensional, optional
If specified, uses this array (or numpy.matrix) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional
A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a numpy.matrix), it will be filled with the appropriate values and returned wrapped in a numpy.matrix object that shares the same memory.

dok_matrix.**todia**(*copy=False*)
Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

dok_matrix.**todok**(*copy=False*)
Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

dok_matrix.**tolil**(*copy=False*)
Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

dok_matrix.**transpose**(*axes=None*, *copy=False*)
Reverses the dimensions of the sparse matrix.

Parameters **axes** : None, optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional
Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p** : *self* with the dimensions reversed.

See also:

np.matrix.transpose
NumPy's implementation of 'transpose' for matrices

dok_matrix.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

dok_matrix.**values**() → list of D's values

dok_matrix.**viewitems**() → a set-like object providing a view on D's items

dok_matrix.**viewkeys**() → a set-like object providing a view on D's keys

dok_matrix.**viewvalues**() → an object providing a view on D's values

class scipy.sparse.**lil_matrix**(*arg1*, *shape=None*, *dtype=None*, *copy=False*)

Row-based linked list sparse matrix

This is a structure for constructing sparse matrices incrementally. Note that inserting a single item can take linear time in the worst case; to construct a matrix efficiently, make sure the items are pre-sorted by index, per row.

This can be instantiated in several ways:

lil_matrix(D) with a dense matrix or rank-2 ndarray D

lil_matrix(S) with another sparse matrix S (equivalent to S.tolil())

lil_matrix((M, N), [dtype]) to construct an empty matrix with shape (M, N) dtype is optional, defaulting to dtype='d'.

Notes

Sparse matrices can be used in arithmetic operations: they support addition, subtraction, multiplication, division, and matrix power.

Advantages of the LIL format

- supports flexible slicing
- changes to the matrix sparsity structure are efficient

Disadvantages of the LIL format

- arithmetic operations LIL + LIL are slow (consider CSR or CSC)
- slow column slicing (consider CSC)
- slow matrix vector products (consider CSR or CSC)

Intended Usage

- LIL is a convenient format for constructing sparse matrices
- once a matrix has been constructed, convert to CSR or CSC format for fast arithmetic and matrix vector operations
- consider using the COO format when constructing large matrices

Data Structure

- An array (`self.rows`) of rows, each of which is a sorted list of column indices of non-zero elements.
- The corresponding nonzero values are stored in similar fashion in `self.data`.

Attributes

shape	Get shape of a matrix.
nnz	Number of stored values, including explicit zeros.

lil_matrix.**shape**
Get shape of a matrix.

lil_matrix.**nnz**
Number of stored values, including explicit zeros.

See also:

count_nonzero
Number of non-zero entries

dtype	(dtype) Data type of the matrix
ndim	(int) Number of dimensions (this is always 2)
data	LIL format data array of the matrix
rows	LIL format row index array of the matrix

Methods

asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of the 'i'th row.
getrowview(i)	Returns a view of the 'i'th row (without copying).
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix
nonzero()	nonzero indices
power(n[, dtype])	Element-wise power.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
toarray([order, out])	See the docstring for *spmatrix.toarray*.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.

Continued on next page

Table 5.162 – continued from previous page

tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.

lil_matrix.**asformat**(*format*)
Return this matrix in a given sparse format

Parameters **format** : {string, None}
desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format
- "csc" for csc_matrix format
- "lil" for lil_matrix format
- "dok" for dok_matrix format and so on

lil_matrix.**asfptype**()
Upcast matrix to a floating point format (if necessary)

lil_matrix.**astype**(*t*)
Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype
Typecode or data-type to which to cast the data.

lil_matrix.**conj**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

lil_matrix.**conjugate**()
Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

lil_matrix.**copy**()
Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

lil_matrix.**count_nonzero**()
Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

lil_matrix.**diagonal**()
Returns the main diagonal of the matrix

lil_matrix.**dot**(*other*)
Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

`lil_matrix.`**`getH`**`()`

Return the Hermitian transpose of this matrix.

See also:

`np.matrix.getH`
NumPy's implementation of *`getH`* for matrices

`lil_matrix.`**`get_shape`**`()`

Get shape of a matrix.

`lil_matrix.`**`getcol`**`(`*j*`)`

Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

`lil_matrix.`**`getformat`**`()`

Format of a matrix representation as a string.

`lil_matrix.`**`getmaxprint`**`()`

Maximum number of elements to display when printed.

`lil_matrix.`**`getnnz`**`(`*axis=None*`)`

Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1
Select between the number of values across the whole matrix, in each column, or in each row.

See also:

`count_nonzero`
Number of non-zero entries

`lil_matrix.`**`getrow`**`(`*i*`)`

Returns a copy of the 'i'th row.

`lil_matrix.`**`getrowview`**`(`*i*`)`

Returns a view of the 'i'th row (without copying).

`lil_matrix.`**`maximum`**`(`*other*`)`

Element-wise maximum between this and another matrix.

`lil_matrix.`**`mean`**`(`*axis=None*, *dtype=None*, *out=None*`)`

Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).

dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.

out : np.matrix, optional

Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

np.matrix.mean
NumPy's implementation of 'mean' for matrices

`lil_matrix.`**`minimum`**`(`*other*`)`
Element-wise minimum between this and another matrix.

`lil_matrix.`**`multiply`**`(`*other*`)`
Point-wise multiplication by another matrix

`lil_matrix.`**`nonzero`**`()`
nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`lil_matrix.`**`power`**`(`*n*, *dtype=None*`)`
Element-wise power.

`lil_matrix.`**`reshape`**`(`*shape*, *order='C'*`)`
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.

order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

np.matrix.reshape
NumPy's implementation of 'reshape' for matrices

`lil_matrix.`**`set_shape`**`(`*shape*`)`
See `reshape`.

`lil_matrix.`**`setdiag`**`(`*values*, *k=0*`)`
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.

k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

`lil_matrix.`**`sum`**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).
dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.
out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix
A matrix with the same shape as *self*, with the specified axis removed.

See also:

`np.matrix.sum`
NumPy's implementation of 'sum' for matrices

`lil_matrix.`**`toarray`**(*order=None*, *out=None*)
See the docstring for *`spmatrix.toarray`*.

`lil_matrix.`**`tobsr`**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

`lil_matrix.`**`tocoo`**(*copy=False*)
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`lil_matrix.`**`tocsc`**(*copy=False*)
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

`lil_matrix.`**`tocsr`**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

`lil_matrix.`**`todense`**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.
out : ndarray, 2-dimensional, optional
If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must

have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional

A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

`lil_matrix.`**`todia`**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

`lil_matrix.`**`todok`**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

`lil_matrix.`**`tolil`**(*copy=False*)

Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`lil_matrix.`**`transpose`**(*axes=None*, *copy=False*)

Reverses the dimensions of the sparse matrix.

Parameters **axes** : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional

Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p** : *self* with the dimensions reversed.

See also:

`np.matrix.transpose`

NumPy's implementation of 'transpose' for matrices

class `scipy.sparse.`**`spmatrix`**(*maxprint=50*)

This class provides a base class for all sparse matrices. It cannot be instantiated. Most of the work is provided by subclasses.

Attributes

nnz	Number of stored values, including explicit zeros.
shape	Get shape of a matrix.

`spmatrix.`**`nnz`**

Number of stored values, including explicit zeros.

See also:

`count_nonzero`

Number of non-zero entries

`spmatrix.`**`shape`**

Get shape of a matrix.

Methods

asformat(format)	Return this matrix in a given sparse format
asfptype()	Upcast matrix to a floating point format (if necessary)
astype(t)	Cast the matrix elements to a specified type.
conj()	Element-wise complex conjugation.
conjugate()	Element-wise complex conjugation.
copy()	Returns a copy of this matrix.
count_nonzero()	Number of non-zero entries, equivalent to
diagonal()	Returns the main diagonal of the matrix
dot(other)	Ordinary dot product
getH()	Return the Hermitian transpose of this matrix.
get_shape()	Get shape of a matrix.
getcol(j)	Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).
getformat()	Format of a matrix representation as a string.
getmaxprint()	Maximum number of elements to display when printed.
getnnz([axis])	Number of stored values, including explicit zeros.
getrow(i)	Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).
maximum(other)	Element-wise maximum between this and another matrix.
mean([axis, dtype, out])	Compute the arithmetic mean along the specified axis.
minimum(other)	Element-wise minimum between this and another matrix.
multiply(other)	Point-wise multiplication by another matrix
nonzero()	nonzero indices
power(n[, dtype])	Element-wise power.
reshape(shape[, order])	Gives a new shape to a sparse matrix without changing its data.
set_shape(shape)	See *reshape*.
setdiag(values[, k])	Set diagonal or off-diagonal elements of the array.
sum([axis, dtype, out])	Sum the matrix elements over a given axis.
toarray([order, out])	Return a dense ndarray representation of this matrix.
tobsr([blocksize, copy])	Convert this matrix to Block Sparse Row format.
tocoo([copy])	Convert this matrix to COOrdinate format.
tocsc([copy])	Convert this matrix to Compressed Sparse Column format.
tocsr([copy])	Convert this matrix to Compressed Sparse Row format.
todense([order, out])	Return a dense matrix representation of this matrix.
todia([copy])	Convert this matrix to sparse DIAgonal format.
todok([copy])	Convert this matrix to Dictionary Of Keys format.
tolil([copy])	Convert this matrix to LInked List format.
transpose([axes, copy])	Reverses the dimensions of the sparse matrix.

spmatrix.**asformat**(*format*)

Return this matrix in a given sparse format

Parameters **format** : {string, None}

desired sparse matrix format

- None for no format conversion
- "csr" for csr_matrix format

- “csc” for csc_matrix format
- “lil” for lil_matrix format
- “dok” for dok_matrix format and so on

`spmatrix.`**`asfptype`**`()`

Upcast matrix to a floating point format (if necessary)

`spmatrix.`**`astype`**`(`*t*`)`

Cast the matrix elements to a specified type.

The data will be copied.

Parameters **t** : string or numpy dtype

Typecode or data-type to which to cast the data.

`spmatrix.`**`conj`**`()`

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`spmatrix.`**`conjugate`**`()`

Element-wise complex conjugation.

If the matrix is of non-complex data type, then this method does nothing and the data is not copied.

`spmatrix.`**`copy`**`()`

Returns a copy of this matrix.

No data/indices will be shared between the returned value and current matrix.

`spmatrix.`**`count_nonzero`**`()`

Number of non-zero entries, equivalent to

np.count_nonzero(a.toarray())

Unlike getnnz() and the nnz property, which return the number of stored entries (the length of the data attribute), this method counts the actual number of non-zero entries in data.

`spmatrix.`**`diagonal`**`()`

Returns the main diagonal of the matrix

`spmatrix.`**`dot`**`(`*other*`)`

Ordinary dot product

Examples

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

`spmatrix.`**`getH`**`()`

Return the Hermitian transpose of this matrix.

See also:

`np.matrix.getH`

NumPy’s implementation of *`getH`* for matrices

`spmatrix.`**`get_shape`**`()`

Get shape of a matrix.

`spmatrix.`**`getcol`**`(`*j*`)`
Returns a copy of column j of the matrix, as an (m x 1) sparse matrix (column vector).

`spmatrix.`**`getformat`**`()`
Format of a matrix representation as a string.

`spmatrix.`**`getmaxprint`**`()`
Maximum number of elements to display when printed.

`spmatrix.`**`getnnz`**`(`*axis=None*`)`
Number of stored values, including explicit zeros.

Parameters **axis** : None, 0, or 1
Select between the number of values across the whole matrix, in each column, or in each row.

See also:

`count_nonzero`
Number of non-zero entries

`spmatrix.`**`getrow`**`(`*i*`)`
Returns a copy of row i of the matrix, as a (1 x n) sparse matrix (row vector).

`spmatrix.`**`maximum`**`(`*other*`)`
Element-wise maximum between this and another matrix.

`spmatrix.`**`mean`**`(`*axis=None*, *dtype=None*, *out=None*`)`
Compute the arithmetic mean along the specified axis.

Returns the average of the matrix elements. The average is taken over all elements in the matrix by default, otherwise over the specified axis. *float64* intermediate and return values are used for integer inputs.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the mean is computed. The default is to compute the mean of all elements in the matrix (i.e. *axis* = *None*).
dtype : data-type, optional
Type to use in computing the mean. For integer inputs, the default is *float64*; for floating point inputs, it is the same as the input dtype.
out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **m** : np.matrix

See also:

`np.matrix.mean`
NumPy's implementation of 'mean' for matrices

`spmatrix.`**`minimum`**`(`*other*`)`
Element-wise minimum between this and another matrix.

`spmatrix.`**`multiply`**`(`*other*`)`
Point-wise multiplication by another matrix

`spmatrix.`**`nonzero`**`()`
nonzero indices

Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the matrix.

Examples

```
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
>>> A.nonzero()
(array([0, 0, 1, 2, 2]), array([0, 1, 2, 0, 2]))
```

`spmatrix.`**`power`**(*n*, *dtype=None*)
Element-wise power.

`spmatrix.`**`reshape`**(*shape*, *order='C'*)
Gives a new shape to a sparse matrix without changing its data.

Parameters **shape** : length-2 tuple of ints
The new shape should be compatible with the original shape.
order : 'C', optional
This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value, as this argument is not used.

Returns **reshaped_matrix** : *self* with the new dimensions of `shape`

See also:

`np.matrix.reshape`
NumPy's implementation of 'reshape' for matrices

`spmatrix.`**`set_shape`**(*shape*)
See `reshape`.

`spmatrix.`**`setdiag`**(*values*, *k=0*)
Set diagonal or off-diagonal elements of the array.

Parameters **values** : array_like
New values of the diagonal elements.
Values may have any length. If the diagonal is longer than values, then the remaining diagonal entries will not be set. If values if longer than the diagonal, then the remaining values are ignored.
If a scalar value is given, all of the diagonal is set to it.
k : int, optional
Which off-diagonal to set, corresponding to elements a[i,i+k]. Default: 0 (the main diagonal).

`spmatrix.`**`sum`**(*axis=None*, *dtype=None*, *out=None*)
Sum the matrix elements over a given axis.

Parameters **axis** : {-2, -1, 0, 1, None} optional
Axis along which the sum is computed. The default is to compute the sum of all the matrix elements, returning a scalar (i.e. *axis* = *None*).
dtype : dtype, optional
The type of the returned matrix and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.
out : np.matrix, optional
Alternative output matrix in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

Returns **sum_along_axis** : np.matrix

A matrix with the same shape as *self*, with the specified axis removed.

See also:

np.matrix.sum
NumPy's implementation of 'sum' for matrices

`spmatrix.`**`toarray`**(*order=None*, *out=None*)
Return a dense ndarray representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.
out : ndarray, 2-dimensional, optional
If specified, uses this array as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method. For most sparse types, *out* is required to be memory contiguous (either C or Fortran ordered).

Returns **arr** : ndarray, 2-dimensional
An array with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed, the same object is returned after being modified in-place to contain the appropriate values.

`spmatrix.`**`tobsr`**(*blocksize=None*, *copy=False*)
Convert this matrix to Block Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant bsr_matrix.

When blocksize=(R, C) is provided, it will be used for construction of the bsr_matrix.

`spmatrix.`**`tocoo`**(*copy=False*)
Convert this matrix to COOrdinate format.

With copy=False, the data/indices may be shared between this matrix and the resultant coo_matrix.

`spmatrix.`**`tocsc`**(*copy=False*)
Convert this matrix to Compressed Sparse Column format.

With copy=False, the data/indices may be shared between this matrix and the resultant csc_matrix.

`spmatrix.`**`tocsr`**(*copy=False*)
Convert this matrix to Compressed Sparse Row format.

With copy=False, the data/indices may be shared between this matrix and the resultant csr_matrix.

`spmatrix.`**`todense`**(*order=None*, *out=None*)
Return a dense matrix representation of this matrix.

Parameters **order** : {'C', 'F'}, optional
Whether to store multi-dimensional data in C (row-major) or Fortran (column-major) order in memory. The default is 'None', indicating the NumPy default of C-ordered. Cannot be specified in conjunction with the *out* argument.
out : ndarray, 2-dimensional, optional
If specified, uses this array (or `numpy.matrix`) as the output buffer instead of allocating a new array to return. The provided array must have the same shape and dtype as the sparse matrix on which you are calling the method.

Returns **arr** : numpy.matrix, 2-dimensional

A NumPy matrix object with the same shape and containing the same data represented by the sparse matrix, with the requested memory order. If *out* was passed and was an array (rather than a `numpy.matrix`), it will be filled with the appropriate values and returned wrapped in a `numpy.matrix` object that shares the same memory.

`spmatrix.`**`todia`**(*copy=False*)

Convert this matrix to sparse DIAgonal format.

With copy=False, the data/indices may be shared between this matrix and the resultant dia_matrix.

`spmatrix.`**`todok`**(*copy=False*)

Convert this matrix to Dictionary Of Keys format.

With copy=False, the data/indices may be shared between this matrix and the resultant dok_matrix.

`spmatrix.`**`tolil`**(*copy=False*)

Convert this matrix to LInked List format.

With copy=False, the data/indices may be shared between this matrix and the resultant lil_matrix.

`spmatrix.`**`transpose`**(*axes=None*, *copy=False*)

Reverses the dimensions of the sparse matrix.

Parameters **axes** : None, optional

This argument is in the signature *solely* for NumPy compatibility reasons. Do not pass in anything except for the default value.

copy : bool, optional

Indicates whether or not attributes of *self* should be copied whenever possible. The degree to which attributes are copied varies depending on the type of sparse matrix being used.

Returns **p** : *self* with the dimensions reversed.

See also:

np.matrix.transpose

NumPy's implementation of 'transpose' for matrices

Functions

Building sparse matrices:

eye(m[, n, k, dtype, format])	Sparse matrix with ones on diagonal
identity(n[, dtype, format])	Identity matrix in sparse format
kron(A, B[, format])	kronecker product of sparse matrices A and B
kronsum(A, B[, format])	kronecker sum of sparse matrices A and B
diags(diagonals[, offsets, shape, format, dtype])	Construct a sparse matrix from diagonals.
spdiags(data, diags, m, n[, format])	Return a sparse matrix from diagonals.
block_diag(mats[, format, dtype])	Build a block diagonal sparse matrix from provided matrices.
tril(A[, k, format])	Return the lower triangular portion of a matrix in sparse format
triu(A[, k, format])	Return the upper triangular portion of a matrix in sparse format
bmat(blocks[, format, dtype])	Build a sparse matrix from sparse sub-blocks
hstack(blocks[, format, dtype])	Stack sparse matrices horizontally (column wise)

Continued on next page

Table 5.165 – continued from previous page

vstack(blocks[, format, dtype])	Stack sparse matrices vertically (row wise)
rand(m, n[, density, format, dtype, ...])	Generate a sparse matrix of the given shape and density with uniformly distributed values.
random(m, n[, density, format, dtype, ...])	Generate a sparse matrix of the given shape and density with randomly distributed values.

scipy.sparse.**eye**(*m*, *n=None*, *k=0*, *dtype=<type 'float'>*, *format=None*)

Sparse matrix with ones on diagonal

Returns a sparse (m x n) matrix where the k-th diagonal is all ones and everything else is zeros.

Parameters

m : int
: Number of rows in the matrix.

n : int, optional
: Number of columns. Default: *m*.

k : int, optional
: Diagonal to place ones on. Default: 0 (main diagonal).

dtype : dtype, optional
: Data type of the matrix.

format : str, optional
: Sparse format of the result, e.g. format="csr", etc.

Examples

```
>>> from scipy import sparse
>>> sparse.eye(3).toarray()
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> sparse.eye(3, dtype=np.int8)
<3x3 sparse matrix of type '<type 'numpy.int8'>'
    with 3 stored elements (1 diagonals) in DIAgonal format>
```

scipy.sparse.**identity**(*n*, *dtype='d'*, *format=None*)

Identity matrix in sparse format

Returns an identity matrix with shape (n,n) using a given sparse format and dtype.

Parameters

n : int
: Shape of the identity matrix.

dtype : dtype, optional
: Data type of the matrix

format : str, optional
: Sparse format of the result, e.g. format="csr", etc.

Examples

```
>>> from scipy.sparse import identity
>>> identity(3).toarray()
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> identity(3, dtype='int8', format='dia')
<3x3 sparse matrix of type '<type 'numpy.int8'>'
        with 3 stored elements (1 diagonals) in DIAgonal format>
```

`scipy.sparse.kron(A, B, format=None)`
kronecker product of sparse matrices A and B

Parameters **A** : sparse or dense matrix
first matrix of the product
B : sparse or dense matrix
second matrix of the product
format : str, optional
format of the result (e.g. "csr")

Returns kronecker product in a sparse matrix format

Examples

```
>>> from scipy import sparse
>>> A = sparse.csr_matrix(np.array([[0, 2], [5, 0]]))
>>> B = sparse.csr_matrix(np.array([[1, 2], [3, 4]]))
>>> sparse.kron(A, B).toarray()
array([[ 0,  0,  2,  4],
       [ 0,  0,  6,  8],
       [ 5, 10,  0,  0],
       [15, 20,  0,  0]])
```

```
>>> sparse.kron(A, [[1, 2], [3, 4]]).toarray()
array([[ 0,  0,  2,  4],
       [ 0,  0,  6,  8],
       [ 5, 10,  0,  0],
       [15, 20,  0,  0]])
```

`scipy.sparse.kronsum(A, B, format=None)`
kronecker sum of sparse matrices A and B

Kronecker sum of two sparse matrices is a sum of two Kronecker products kron(I_n,A) + kron(B,I_m) where A has shape (m,m) and B has shape (n,n) and I_m and I_n are identity matrices of shape (m,m) and (n,n) respectively.

Parameters **A**
square matrix
B
square matrix
format : str
format of the result (e.g. "csr")

Returns kronecker sum in a sparse matrix format

`scipy.sparse.diags(diagonals, offsets=0, shape=None, format=None, dtype=None)`
Construct a sparse matrix from diagonals.

Parameters **diagonals** : sequence of array_like
Sequence of arrays containing the matrix diagonals, corresponding to *offsets*.
offsets : sequence of int or an int, optional
Diagonals to set:

- k = 0 the main diagonal (default)
- k > 0 the k-th upper diagonal
- k < 0 the k-th lower diagonal

shape : tuple of int, optional
Shape of the result. If omitted, a square matrix large enough to contain the diagonals is returned.

format : {"dia", "csr", "csc", "lil", ...}, optional
Matrix format of the result. By default (format=None) an appropriate sparse matrix format is returned. This choice is subject to change.

dtype : dtype, optional
Data type of the matrix.

See also:

spdiags construct matrix from diagonals

Notes

This function differs from *spdiags* in the way it handles off-diagonals.

The result from *diags* is the sparse equivalent of:

```
np.diag(diagonals[0], offsets[0])
+ ...
+ np.diag(diagonals[k], offsets[k])
```

Repeated diagonal offsets are disallowed.

New in version 0.11.

Examples

```
>>> from scipy.sparse import diags
>>> diagonals = [[1, 2, 3, 4], [1, 2, 3], [1, 2]]
>>> diags(diagonals, [0, -1, 2]).toarray()
array([[1, 0, 1, 0],
       [1, 2, 0, 2],
       [0, 2, 3, 0],
       [0, 0, 3, 4]])
```

Broadcasting of scalars is supported (but shape needs to be specified):

```
>>> diags([1, -2, 1], [-1, 0, 1], shape=(4, 4)).toarray()
array([[-2.,  1.,  0.,  0.],
       [ 1., -2.,  1.,  0.],
       [ 0.,  1., -2.,  1.],
       [ 0.,  0.,  1., -2.]])
```

If only one diagonal is wanted (as in `numpy.diag`), the following works as well:

```
>>> diags([1, 2, 3], 1).toarray()
array([[ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  2.,  0.],
       [ 0.,  0.,  0.,  3.],
       [ 0.,  0.,  0.,  0.]])
```

scipy.sparse.**spdiags**(*data*, *diags*, *m*, *n*, *format=None*)
Return a sparse matrix from diagonals.

Parameters **data** : array_like
matrix diagonals stored row-wise

diags : diagonals to set

- k = 0 the main diagonal
- k > 0 the k-th upper diagonal
- k < 0 the k-th lower diagonal

m, n : int
shape of the result

format : str, optional

Format of the result. By default (format=None) an appropriate sparse matrix format is returned. This choice is subject to change.

See also:

diags more convenient form of this function

dia_matrix the sparse DIAgonal format.

Examples

```
>>> from scipy.sparse import spdiags
>>> data = np.array([[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]])
>>> diags = np.array([0, -1, 2])
>>> spdiags(data, diags, 4, 4).toarray()
array([[1, 0, 3, 0],
       [1, 2, 0, 4],
       [0, 2, 3, 0],
       [0, 0, 3, 4]])
```

`scipy.sparse.`**`block_diag`**(*mats, format=None, dtype=None*)

Build a block diagonal sparse matrix from provided matrices.

Parameters **mats** : sequence of matrices

Input matrices.

format : str, optional

The sparse format of the result (e.g. "csr"). If not given, the matrix is returned in "coo" format.

dtype : dtype specifier, optional

The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

Returns **res** : sparse matrix

See also:

bmat, *diags*

Notes

New in version 0.11.0.

Examples

```
>>> from scipy.sparse import coo_matrix, block_diag
>>> A = coo_matrix([[1, 2], [3, 4]])
>>> B = coo_matrix([[5], [6]])
>>> C = coo_matrix([[7]])
>>> block_diag((A, B, C)).toarray()
array([[1, 2, 0, 0],
       [3, 4, 0, 0],
       [0, 0, 5, 0],
       [0, 0, 6, 0],
       [0, 0, 0, 7]])
```

`scipy.sparse.`**`tril`**(*A, k=0, format=None*)

Return the lower triangular portion of a matrix in sparse format

Returns the elements on or below the k-th diagonal of the matrix A.

- k = 0 corresponds to the main diagonal
- k > 0 is above the main diagonal

- k < 0 is below the main diagonal

Parameters **A** : dense or sparse matrix
Matrix whose lower triangular portion is desired.
k : integer
The top-most diagonal of the lower triangle.
format : string
Sparse format of the result, e.g. format="csr", etc.

Returns **L** : sparse matrix
Lower triangular portion of A in sparse format.

See also:

triu upper triangle in sparse format

Examples

```
>>> from scipy.sparse import csr_matrix, tril
>>> A = csr_matrix([[1, 2, 0, 0, 3], [4, 5, 0, 6, 7], [0, 0, 8, 9, 0]],
...                dtype='int32')
>>> A.toarray()
array([[1, 2, 0, 0, 3],
       [4, 5, 0, 6, 7],
       [0, 0, 8, 9, 0]])
>>> tril(A).toarray()
array([[1, 0, 0, 0, 0],
       [4, 5, 0, 0, 0],
       [0, 0, 8, 0, 0]])
>>> tril(A).nnz
4
>>> tril(A, k=1).toarray()
array([[1, 2, 0, 0, 0],
       [4, 5, 0, 0, 0],
       [0, 0, 8, 9, 0]])
>>> tril(A, k=-1).toarray()
array([[0, 0, 0, 0, 0],
       [4, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> tril(A, format='csc')
<3x5 sparse matrix of type '<type 'numpy.int32'>'
        with 4 stored elements in Compressed Sparse Column format>
```

`scipy.sparse.`**`triu`**(*A*, *k=0*, *format=None*)

Return the upper triangular portion of a matrix in sparse format

Returns the elements on or above the k-th diagonal of the matrix A.

- k = 0 corresponds to the main diagonal
- k > 0 is above the main diagonal
- k < 0 is below the main diagonal

Parameters **A** : dense or sparse matrix
Matrix whose upper triangular portion is desired.
k : integer
The bottom-most diagonal of the upper triangle.
format : string
Sparse format of the result, e.g. format="csr", etc.

Returns **L** : sparse matrix
Upper triangular portion of A in sparse format.

See also:

tril lower triangle in sparse format

Examples

```
>>> from scipy.sparse import csr_matrix, triu
>>> A = csr_matrix([[1, 2, 0, 0, 3], [4, 5, 0, 6, 7], [0, 0, 8, 9, 0]],
...                dtype='int32')
>>> A.toarray()
array([[1, 2, 0, 0, 3],
       [4, 5, 0, 6, 7],
       [0, 0, 8, 9, 0]])
>>> triu(A).toarray()
array([[1, 2, 0, 0, 3],
       [0, 5, 0, 6, 7],
       [0, 0, 8, 9, 0]])
>>> triu(A).nnz
8
>>> triu(A, k=1).toarray()
array([[0, 2, 0, 0, 3],
       [0, 0, 0, 6, 7],
       [0, 0, 0, 9, 0]])
>>> triu(A, k=-1).toarray()
array([[1, 2, 0, 0, 3],
       [4, 5, 0, 6, 7],
       [0, 0, 8, 9, 0]])
>>> triu(A, format='csc')
<3x5 sparse matrix of type '<type 'numpy.int32'>'
        with 8 stored elements in Compressed Sparse Column format>
```

scipy.sparse.**bmat**(*blocks*, *format=None*, *dtype=None*)

Build a sparse matrix from sparse sub-blocks

Parameters **blocks** : array_like

Grid of sparse matrices with compatible shapes. An entry of None implies an all-zero matrix.

format : {'bsr', 'coo', 'csc', 'csr', 'dia', 'dok', 'lil'}, optional

The sparse format of the result (e.g. "csr"). By default an appropriate sparse matrix format is returned. This choice is subject to change.

dtype : dtype, optional

The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

Returns **bmat** : sparse matrix

See also:

block_diag, *diags*

Examples

```
>>> from scipy.sparse import coo_matrix, bmat
>>> A = coo_matrix([[1, 2], [3, 4]])
>>> B = coo_matrix([[5], [6]])
>>> C = coo_matrix([[7]])
>>> bmat([[A, B], [None, C]]).toarray()
array([[1, 2, 5],
       [3, 4, 6],
       [0, 0, 7]])
```

```
>>> bmat([[A, None], [None, C]]).toarray()
array([[1, 2, 0],
       [3, 4, 0],
       [0, 0, 7]])
```

`scipy.sparse.hstack`(*blocks*, *format=None*, *dtype=None*)
Stack sparse matrices horizontally (column wise)

Parameters **blocks**
sequence of sparse matrices with compatible shapes

format : str
sparse format of the result (e.g. "csr") by default an appropriate sparse matrix format is returned. This choice is subject to change.

dtype : dtype, optional
The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

See also:

vstack stack sparse matrices vertically (row wise)

Examples

```
>>> from scipy.sparse import coo_matrix, hstack
>>> A = coo_matrix([[1, 2], [3, 4]])
>>> B = coo_matrix([[5], [6]])
>>> hstack([A,B]).toarray()
array([[1, 2, 5],
       [3, 4, 6]])
```

`scipy.sparse.vstack`(*blocks*, *format=None*, *dtype=None*)
Stack sparse matrices vertically (row wise)

Parameters **blocks**
sequence of sparse matrices with compatible shapes

format : str, optional
sparse format of the result (e.g. "csr") by default an appropriate sparse matrix format is returned. This choice is subject to change.

dtype : dtype, optional
The data-type of the output matrix. If not given, the dtype is determined from that of *blocks*.

See also:

hstack stack sparse matrices horizontally (column wise)

Examples

```
>>> from scipy.sparse import coo_matrix, vstack
>>> A = coo_matrix([[1, 2], [3, 4]])
>>> B = coo_matrix([[5, 6]])
>>> vstack([A, B]).toarray()
array([[1, 2],
       [3, 4],
       [5, 6]])
```

`scipy.sparse.rand`(*m*, *n*, *density=0.01*, *format='coo'*, *dtype=None*, *random_state=None*)
Generate a sparse matrix of the given shape and density with uniformly distributed values.

Parameters **m, n** : int
shape of the matrix
density : real, optional
density of the generated matrix: density equal to one means a full matrix, density of 0 means a matrix with no non-zero items.
format : str, optional
sparse matrix format.
dtype : dtype, optional
type of the returned matrix values.
random_state : {numpy.random.RandomState, int}, optional
Random number generator or random seed. If not given, the singleton numpy.random will be used.

Notes

Only float types are supported for now.

`scipy.sparse.`**`random`**(*m*, *n*, *density=0.01*, *format='coo'*, *dtype=None*, *random_state=None*, *data_rvs=None*)

Generate a sparse matrix of the given shape and density with randomly distributed values.

Parameters **m, n** : int
shape of the matrix
density : real, optional
density of the generated matrix: density equal to one means a full matrix, density of 0 means a matrix with no non-zero items.
format : str, optional
sparse matrix format.
dtype : dtype, optional
type of the returned matrix values.
random_state : {numpy.random.RandomState, int}, optional
Random number generator or random seed. If not given, the singleton numpy.random will be used. This random state will be used for sampling the sparsity structure, but not necessarily for sampling the values of the structurally nonzero entries of the matrix.
data_rvs : callable, optional
Samples a requested number of random values. This function should take a single argument specifying the length of the ndarray that it will return. The structurally nonzero entries of the sparse random matrix will be taken from the array sampled by this function. By default, uniform [0, 1) random values will be sampled using the same random state as is used for sampling the sparsity structure.

Notes

Only float types are supported for now.

Examples

```
>>> from scipy.sparse import random
>>> from scipy import stats
>>> class CustomRandomState(object):
...     def randint(self, k):
...         i = np.random.randint(k)
...         return i - i % 2
>>> rs = CustomRandomState()
>>> rvs = stats.poisson(25, loc=10).rvs
```

```
>>> S = random(3, 4, density=0.25, random_state=rs, data_rvs=rvs)
>>> S.A
array([[ 36.,    0.,  33.,    0.],   # random
       [  0.,    0.,   0.,    0.],
       [  0.,    0.,  36.,    0.]])
```

Save and load sparse matrices:

save_npz(file, matrix[, compressed])	Save a sparse matrix to a file using .npz format.
load_npz(file)	Load a sparse matrix from a file using .npz format.

scipy.sparse.**save_npz**(*file*, *matrix*, *compressed=True*)

Save a sparse matrix to a file using .npz format.

Parameters **file** : str or file-like object

Either the file name (string) or an open file (file-like object) where the data will be saved. If file is a string, the .npz extension will be appended to the file name if it is not already there.

matrix: spmatrix (format: ''csc'', ''csr'', ''bsr'', ''dia'' or coo'')

The sparse matrix to save.

compressed : bool, optional

Allow compressing the file. Default: True

See also:

scipy.sparse.load_npz

Load a sparse matrix from a file using .npz format.

numpy.savez

Save several arrays into a .npz archive.

numpy.savez_compressed

Save several arrays into a compressed .npz archive.

Examples

Store sparse matrix to disk, and load it again:

```
>>> import scipy.sparse
>>> sparse_matrix = scipy.sparse.csc_matrix(np.array([[0, 0, 3], [4, 0, 0]]))
>>> sparse_matrix
<2x3 sparse matrix of type '<type 'numpy.int64'>'
   with 2 stored elements in Compressed Sparse Column format>
>>> sparse_matrix.todense()
matrix([[0, 0, 3],
        [4, 0, 0]], dtype=int64)
```

```
>>> scipy.sparse.save_npz('/tmp/sparse_matrix.npz', sparse_matrix)
>>> sparse_matrix = scipy.sparse.load_npz('/tmp/sparse_matrix.npz')
```

```
>>> sparse_matrix
<2x3 sparse matrix of type '<type 'numpy.int64'>'
   with 2 stored elements in Compressed Sparse Column format>
>>> sparse_matrix.todense()
matrix([[0, 0, 3],
        [4, 0, 0]], dtype=int64)
```

`scipy.sparse.load_npz(file)`

Load a sparse matrix from a file using `.npz` format.

Parameters **file** : str or file-like object
Either the file name (string) or an open file (file-like object) where the data will be loaded.

Returns **result** : csc_matrix, csr_matrix, bsr_matrix, dia_matrix or coo_matrix
A sparse matrix containing the loaded data.

Raises **IOError**
If the input file does not exist or cannot be read.

See also:

scipy.sparse.save_npz Save a sparse matrix to a file using `.npz` format.

numpy.load Load several arrays from a `.npz` archive.

Sparse matrix tools:

find(A)	Return the indices and values of the nonzero elements of a matrix

`scipy.sparse.find(A)`

Return the indices and values of the nonzero elements of a matrix

Parameters **A** : dense or sparse matrix
Matrix whose nonzero elements are desired.

Returns **(I,J,V)** : tuple of arrays
I,J, and V contain the row indices, column indices, and values of the nonzero matrix entries.

Examples

```
>>> from scipy.sparse import csr_matrix, find
>>> A = csr_matrix([[7.0, 8.0, 0],[0, 0, 9.0]])
>>> find(A)
(array([0, 0, 1], dtype=int32), array([0, 1, 2], dtype=int32), array([ 7.,  8., 
→9.]))
```

Identifying sparse matrices:

issparse(x)
isspmatrix(x)
isspmatrix_csc(x)
isspmatrix_csr(x)
isspmatrix_bsr(x)
isspmatrix_lil(x)
isspmatrix_dok(x)
isspmatrix_coo(x)
isspmatrix_dia(x)

`scipy.sparse.issparse(x)`

`scipy.sparse.isspmatrix(x)`

`scipy.sparse.isspmatrix_csc(x)`

scipy.sparse.**isspmatrix_csr**(*x*)

scipy.sparse.**isspmatrix_bsr**(*x*)

scipy.sparse.**isspmatrix_lil**(*x*)

scipy.sparse.**isspmatrix_dok**(*x*)

scipy.sparse.**isspmatrix_coo**(*x*)

scipy.sparse.**isspmatrix_dia**(*x*)

Submodules

csgraph
linalg

Compressed Sparse Graph Routines (scipy.sparse.csgraph)

Fast graph algorithms based on sparse matrix representations.

Contents

connected_components(csgraph[, directed, ...])	Analyze the connected components of a sparse graph
laplacian(csgraph[, normed, return_diag, ...])	Return the Laplacian matrix of a directed graph.
shortest_path(csgraph[, method, directed, ...])	Perform a shortest-path graph search on a positive directed or undirected graph.
dijkstra(csgraph[, directed, indices, ...])	Dijkstra algorithm using Fibonacci Heaps
floyd_warshall(csgraph[, directed, ...])	Compute the shortest path lengths using the Floyd-Warshall algorithm
bellman_ford(csgraph[, directed, indices, ...])	Compute the shortest path lengths using the Bellman-Ford algorithm.
johnson(csgraph[, directed, indices, ...])	Compute the shortest path lengths using Johnson's algorithm.
breadth_first_order(csgraph, i_start[, ...])	Return a breadth-first ordering starting with specified node.
depth_first_order(csgraph, i_start[, ...])	Return a depth-first ordering starting with specified node.
breadth_first_tree(csgraph, i_start[, directed])	Return the tree generated by a breadth-first search
depth_first_tree(csgraph, i_start[, directed])	Return a tree generated by a depth-first search.
minimum_spanning_tree(csgraph[, overwrite])	Return a minimum spanning tree of an undirected graph
reverse_cuthill_mckee(graph[, symmetric_mode])	Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering.
maximum_bipartite_matching(graph[, perm_type])	Returns an array of row or column permutations that makes the diagonal of a nonsingular square CSC sparse matrix zero free.

Continued on next page

Table 5.170 – continued from previous page

structural_rank(graph)	Compute the structural rank of a graph (matrix) with a given sparsity pattern.
NegativeCycleError	

scipy.sparse.csgraph.**connected_components**(*csgraph*, *directed=True*, *connection='weak'*, *return_labels=True*)

Analyze the connected components of a sparse graph

New in version 0.11.0.

Parameters

csgraph : array_like or sparse matrix
The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

directed : bool, optional
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

connection : str, optional
['weak'|'strong']. For directed graphs, the type of connection to use. Nodes i and j are strongly connected if a path exists both from i to j and from j to i. Nodes i and j are weakly connected if only one of these paths exists. If directed == False, this keyword is not referenced.

return_labels : bool, optional
If True (default), then return the labels for each of the connected components.

Returns

n_components: int
The number of connected components.

labels: ndarray
The length-N array of labels of the connected components.

References

[R14]

scipy.sparse.csgraph.**laplacian**(*csgraph*, *normed=False*, *return_diag=False*, *use_out_degree=False*)

Return the Laplacian matrix of a directed graph.

Parameters

csgraph : array_like or sparse matrix, 2 dimensions
compressed-sparse graph, with shape (N, N).

normed : bool, optional
If True, then compute normalized Laplacian.

return_diag : bool, optional
If True, then also return an array related to vertex degrees.

use_out_degree : bool, optional
If True, then use out-degree instead of in-degree. This distinction matters only if the graph is asymmetric. Default: False.

Returns

lap : ndarray or sparse matrix
The N x N laplacian matrix of csgraph. It will be a numpy array (dense) if the input was dense, or a sparse matrix otherwise.

diag : ndarray, optional
The length-N diagonal of the Laplacian matrix. For the normalized Laplacian, this is the array of square roots of vertex degrees or 1 if the degree is zero.

Notes

The Laplacian matrix of a graph is sometimes referred to as the "Kirchoff matrix" or the "admittance matrix", and is useful in many parts of spectral graph theory. In particular, the eigen-decomposition of the laplacian matrix can give insight into many properties of the graph.

Examples

```
>>> from scipy.sparse import csgraph
>>> G = np.arange(5) * np.arange(5)[:, np.newaxis]
>>> G
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12],
       [ 0,  4,  8, 12, 16]])
>>> csgraph.laplacian(G, normed=False)
array([[  0,   0,   0,   0,   0],
       [  0,   9,  -2,  -3,  -4],
       [  0,  -2,  16,  -6,  -8],
       [  0,  -3,  -6,  21, -12],
       [  0,  -4,  -8, -12,  24]])
```

`scipy.sparse.csgraph.`**`shortest_path`**(*csgraph*, *method='auto'*, *directed=True*, *return_predecessors=False*, *unweighted=False*, *overwrite=False*, *indices=None*)

Perform a shortest-path graph search on a positive directed or undirected graph.

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions

The N x N array of distances representing the input graph.

method : string ['auto'|'FW'|'D'], optional

Algorithm to use for shortest paths. Options are:

'auto' – (default) select the best among 'FW', 'D', 'BF', or 'J'

based on the input data.

'FW' – Floyd-Warshall algorithm. Computational cost is

approximately `O[N^3]`. The input csgraph will be converted to a dense representation.

'D' – Dijkstra's algorithm with Fibonacci heaps. Computational

cost is approximately `O[N(N*k + N*log(N))]`, where `k` is the average number of connected edges per node. The input csgraph will be converted to a csr representation.

'BF' – Bellman-Ford algorithm. This algorithm can be used when

weights are negative. If a negative cycle is encountered, an error will be raised. Computational cost is approximately `O[N(N^2 k)]`, where `k` is the average number of connected edges per node. The input csgraph will be converted to a csr representation.

'J' – Johnson's algorithm. Like the Bellman-Ford algorithm,

Johnson's algorithm is designed for use when the weights are negative. It combines the Bellman-Ford algorithm with Dijkstra's algorithm for faster computation.

directed : bool, optional
: If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

return_predecessors : bool, optional
: If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
: If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

overwrite : bool, optional
: If True, overwrite csgraph with the result. This applies only if method == 'FW' and csgraph is a dense, c-ordered array with dtype=float64.

indices : array_like or int, optional
: If specified, only compute the paths for the points at the given indices. Incompatible with method == 'FW'.

Returns

dist_matrix : ndarray
: The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
: Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises

NegativeCycleError:
: if there are negative cycles in the graph

Notes

As currently implemented, Dijkstra's algorithm and Johnson's algorithm do not work for graphs with direction-dependent distances when directed == False. i.e., if csgraph[i,j] and csgraph[j,i] are non-equal edges, method='D' may yield an incorrect result.

`scipy.sparse.csgraph.dijkstra`(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*, *limit=np.inf*)

Dijkstra algorithm using Fibonacci Heaps

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions
: The N x N array of non-negative distances representing the input graph.

directed : bool, optional
: If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional
: if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional
: If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

limit : float, optional
The maximum distance to calculate, must be >= 0. Using a smaller limit will decrease computation time by aborting calculations between pairs that are separated by a distance > limit. For such pairs, the distance will be equal to np.inf (i.e., not connected).
New in version 0.14.0.

Returns **dist_matrix** : ndarray
The matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Notes

As currently implemented, Dijkstra's algorithm does not work for graphs with direction-dependent distances when directed == False. i.e., if csgraph[i,j] and csgraph[j,i] are not equal and both are nonzero, setting directed=False will not yield the correct result.

Also, this routine does not work for graphs with negative distances. Negative distances can lead to infinite cycles that must be handled by specialized algorithms such as Bellman-Ford's algorithm or Johnson's algorithm.

`scipy.sparse.csgraph.`**`floyd_warshall`**(*csgraph*, *directed=True*, *return_predecessors=False*, *unweighted=False*, *overwrite=False*)

Compute the shortest path lengths using the Floyd-Warshall algorithm

New in version 0.11.0.

Parameters **csgraph** : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

overwrite : bool, optional
If True, overwrite csgraph with the result. This applies only if csgraph is a dense, c-ordered array with dtype=float64.

Returns **dist_matrix** : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the

predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises **NegativeCycleError:**
if there are negative cycles in the graph

`scipy.sparse.csgraph.`**`bellman_ford`**(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*)

Compute the shortest path lengths using the Bellman-Ford algorithm.

The Bellman-ford algorithm can robustly deal with graphs with negative weights. If a negative cycle is detected, an error is raised. For graphs without negative edge weights, dijkstra's algorithm may be faster.

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional
if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

dist_matrix : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises **NegativeCycleError:**
if there are negative cycles in the graph

Notes

This routine is specially designed for graphs with negative edge weights. If all edge weights are positive, then Dijkstra's algorithm is a better choice.

`scipy.sparse.csgraph.`**`johnson`**(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*)

Compute the shortest path lengths using Johnson's algorithm.

Johnson's algorithm combines the Bellman-Ford algorithm and Dijkstra's algorithm to quickly find shortest paths in a way that is robust to the presence of negative cycles. If a negative cycle is detected, an error is raised. For graphs without negative edge weights, dijkstra() may be faster.

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions

The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional
if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns

dist_matrix : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises

NegativeCycleError:
if there are negative cycles in the graph

Notes

This routine is specially designed for graphs with negative edge weights. If all edge weights are positive, then Dijkstra's algorithm is a better choice.

`scipy.sparse.csgraph.`**`breadth_first_order`**(*csgraph*, *i_start*, *directed=True*, *return_predecessors=True*)

Return a breadth-first ordering starting with specified node.

Note that a breadth-first order is not unique, but the tree which it generates is unique.

New in version 0.11.0.

Parameters

csgraph : array_like or sparse matrix
The N x N compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int
The index of starting node.

directed : bool, optional
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

return_predecessors : bool, optional
If True (default), then return the predecesor array (see below).

Returns

node_array : ndarray, one dimension
The breadth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.

predecessors : ndarray, one dimension
Returned only if return_predecessors is True. The length-N list of predecessors of each node in a breadth-first tree. If node i is in the tree, then its

parent is given by predecessors[i]. If node i is not in the tree (and for the parent node) then predecessors[i] = -9999.

`scipy.sparse.csgraph.`**`depth_first_order`**(*csgraph*, *i_start*, *directed=True*, *return_predecessors=True*)

Return a depth-first ordering starting with specified node.

Note that a depth-first order is not unique. Furthermore, for graphs with cycles, the tree generated by a depth-first search is not unique either.

New in version 0.11.0.

Parameters

csgraph : array_like or sparse matrix
The N x N compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int
The index of starting node.

directed : bool, optional
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

return_predecessors : bool, optional
If True (default), then return the predecesor array (see below).

Returns

node_array : ndarray, one dimension
The breadth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.

predecessors : ndarray, one dimension
Returned only if return_predecessors is True. The length-N list of predecessors of each node in a breadth-first tree. If node i is in the tree, then its parent is given by predecessors[i]. If node i is not in the tree (and for the parent node) then predecessors[i] = -9999.

`scipy.sparse.csgraph.`**`breadth_first_tree`**(*csgraph*, *i_start*, *directed=True*)

Return the tree generated by a breadth-first search

Note that a breadth-first tree from a specified node is unique.

New in version 0.11.0.

Parameters

csgraph : array_like or sparse matrix
The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int
The index of starting node.

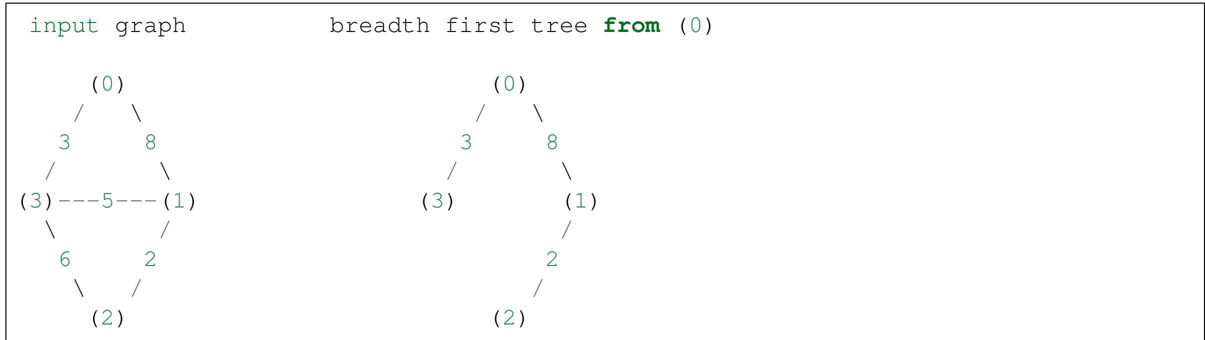
directed : bool, optional
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns

cstree : csr matrix
The N x N directed compressed-sparse representation of the breadth- first tree drawn from csgraph, starting at the specified node.

Examples

The following example shows the computation of a depth-first tree over a simple four-component graph, starting at node 0:



In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import breadth_first_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = breadth_first_tree(X, 0, directed=False)
>>> Tcsr.toarray().astype(int)
array([[0, 8, 0, 3],
       [0, 0, 2, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Note that the resulting graph is a Directed Acyclic Graph which spans the graph. A breadth-first tree from a given node is unique.

`scipy.sparse.csgraph.`**`depth_first_tree`**(*csgraph*, *i_start*, *directed=True*)

Return a tree generated by a depth-first search.

Note that a tree generated by a depth-first search is not unique: it depends on the order that the children of each node are searched.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int

The index of starting node.

directed : bool, optional

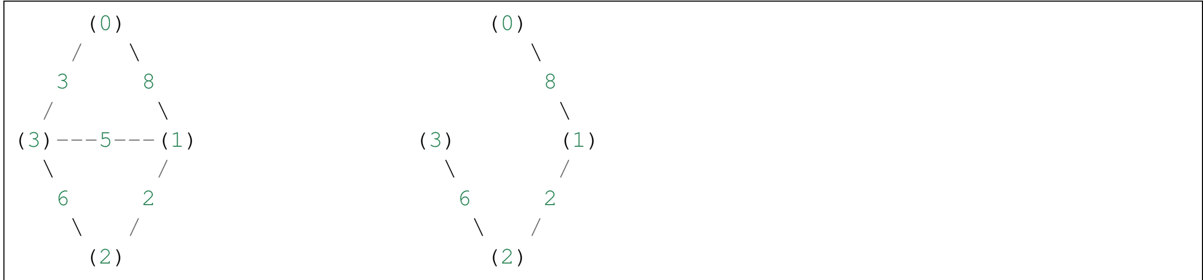
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns **cstree** : csr matrix

The N x N directed compressed-sparse representation of the depth- first tree drawn from csgraph, starting at the specified node.

Examples

The following example shows the computation of a depth-first tree over a simple four-component graph, starting at node 0:



In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import depth_first_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = depth_first_tree(X, 0, directed=False)
>>> Tcsr.toarray().astype(int)
array([[0, 8, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 6],
       [0, 0, 0, 0]])
```

Note that the resulting graph is a Directed Acyclic Graph which spans the graph. Unlike a breadth-first tree, a depth-first tree of a given graph is not unique if the graph contains cycles. If the above solution had begun with the edge connecting nodes 0 and 3, the result would have been different.

`scipy.sparse.csgraph.`**`minimum_spanning_tree`**(*csgraph*, *overwrite=False*)

Return a minimum spanning tree of an undirected graph

A minimum spanning tree is a graph consisting of the subset of edges which together connect all connected nodes, while minimizing the total sum of weights on the edges. This is computed using the Kruskal algorithm.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix, 2 dimensions

The N x N matrix representing an undirected graph over N nodes (see notes below).

overwrite : bool, optional

if true, then parts of the input graph will be overwritten for efficiency.

Returns **span_tree** : csr matrix

The N x N compressed-sparse representation of the undirected minimum spanning tree over the input (see notes below).

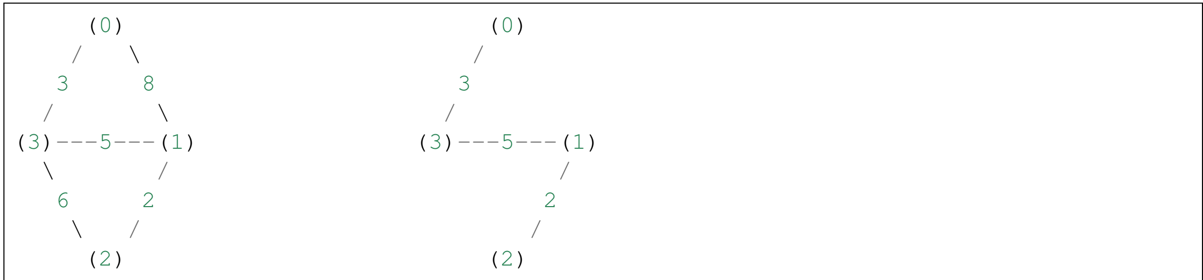
Notes

This routine uses undirected graphs as input and output. That is, if graph[i, j] and graph[j, i] are both zero, then nodes i and j do not have an edge connecting them. If either is nonzero, then the two are connected by the minimum nonzero value of the two.

Examples

The following example shows the computation of a minimum spanning tree over a simple four-component graph:

```
input graph             minimum spanning tree
```



```
      (0)                         (0)
     /   \                       /
    3     8                     3
   /       \                   /
(3)---5---(1)               (3)---5---(1)
   \       /                           /
    6     2                           2
     \   /                           /
      (2)                         (2)
```

It is easy to see from inspection that the minimum spanning tree involves removing the edges with weights 8 and 6. In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import minimum_spanning_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = minimum_spanning_tree(X)
>>> Tcsr.toarray().astype(int)
array([[0, 0, 0, 3],
       [0, 0, 2, 5],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

`scipy.sparse.csgraph.`**`reverse_cuthill_mckee`**(*graph*, *symmetric_mode=False*)

Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering.

It is assumed by default, `symmetric_mode=False`, that the input matrix is not symmetric and works on the matrix `A+A.T`. If you are guaranteed that the matrix is symmetric in structure (values of matrix elements do not matter) then set `symmetric_mode=True`.

Parameters **graph** : sparse matrix

Input sparse in CSC or CSR sparse matrix format.

symmetric_mode : bool, optional

Is input matrix guaranteed to be symmetric.

Returns **perm** : ndarray

Array of permuted row and column indices.

Notes

New in version 0.15.0.

References

E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", ACM '69 Proceedings of the 1969 24th national conference, (1969).

`scipy.sparse.csgraph.`**`maximum_bipartite_matching`**(*graph*, *perm_type='row'*)

Returns an array of row or column permutations that makes the diagonal of a nonsingular square CSC sparse matrix zero free.

Such a permutation is always possible provided that the matrix is nonsingular. This function looks at the structure of the matrix only. The input matrix will be converted to CSC matrix format if necessary.

Parameters **graph** : sparse matrix

Input sparse in CSC format

perm_type : str, {'row', 'column'}

Type of permutation to generate.

Returns **perm** : ndarray
Array of row or column permutations.

Notes

This function relies on a maximum cardinality bipartite matching algorithm based on a breadth-first search (BFS) of the underlying graph.

New in version 0.15.0.

References

I. S. Duff, K. Kaya, and B. Ucar, "Design, Implementation, and Analysis of Maximum Transversal Algorithms", ACM Trans. Math. Softw. 38, no. 2, (2011).

`scipy.sparse.csgraph.`**`structural_rank`**(*graph*)

Compute the structural rank of a graph (matrix) with a given sparsity pattern.

The structural rank of a matrix is the number of entries in the maximum transversal of the corresponding bipartite graph, and is an upper bound on the numerical rank of the matrix. A graph has full structural rank if it is possible to permute the elements to make the diagonal zero-free.

Parameters **graph** : sparse matrix
Input sparse matrix.

Returns **rank** : int
The structural rank of the sparse graph.

New in version 0.19.0.

References

[R15], *[R16]*

exception `scipy.sparse.csgraph.`**`NegativeCycleError`**

construct_dist_matrix(graph, predecessors[, ...])	Construct distance matrix from a predecessor matrix
csgraph_from_dense(graph[, null_value, ...])	Construct a CSR-format sparse graph from a dense matrix.
csgraph_from_masked(graph)	Construct a CSR-format graph from a masked array.
csgraph_masked_from_dense(graph[, ...])	Construct a masked array graph representation from a dense matrix.
csgraph_to_dense(csgraph[, null_value])	Convert a sparse graph representation to a dense representation
csgraph_to_masked(csgraph)	Convert a sparse graph representation to a masked array representation
reconstruct_path(csgraph, predecessors[, ...])	Construct a tree from a graph and a predecessor list.

`scipy.sparse.csgraph.`**`construct_dist_matrix`**(*graph*, *predecessors*, *directed=True*, *null_value=np.inf*)

Construct distance matrix from a predecessor matrix

New in version 0.11.0.

Parameters **graph** : array_like or sparse
The N x N matrix representation of a directed or undirected graph. If dense, then non-edges are indicated by zeros or infinities.

predecessors : array_like
The N x N matrix of predecessors of each node (see Notes below).

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then operate on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

null_value : bool, optional

value to use for distances between unconnected nodes. Default is np.inf

Returns **dist_matrix** : ndarray

The N x N matrix of distances between nodes along the path specified by the predecessor matrix. If no path exists, the distance is zero.

Notes

The predecessor matrix is of the form returned by `graph_shortest_path`. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

`scipy.sparse.csgraph.`**`csgraph_from_dense`**(*graph*, *null_value=0*, *nan_null=True*, *infinity_null=True*)

Construct a CSR-format sparse graph from a dense matrix.

New in version 0.11.0.

Parameters **graph** : array_like

Input graph. Shape should be (n_nodes, n_nodes).

null_value : float or None (optional)

Value that denotes non-edges in the graph. Default is zero.

infinity_null : bool

If True (default), then infinite entries (both positive and negative) are treated as null edges.

nan_null : bool

If True (default), then NaN entries are treated as non-edges

Returns **csgraph** : csr_matrix

Compressed sparse representation of graph,

`scipy.sparse.csgraph.`**`csgraph_from_masked`**(*graph*)

Construct a CSR-format graph from a masked array.

New in version 0.11.0.

Parameters **graph** : MaskedArray

Input graph. Shape should be (n_nodes, n_nodes).

Returns **csgraph** : csr_matrix

Compressed sparse representation of graph,

`scipy.sparse.csgraph.`**`csgraph_masked_from_dense`**(*graph*, *null_value=0*, *nan_null=True*, *infinity_null=True*, *copy=True*)

Construct a masked array graph representation from a dense matrix.

New in version 0.11.0.

Parameters **graph** : array_like

Input graph. Shape should be (n_nodes, n_nodes).

null_value : float or None (optional)

Value that denotes non-edges in the graph. Default is zero.

infinity_null : bool

If True (default), then infinite entries (both positive and negative) are treated as null edges.

nan_null : bool

If True (default), then NaN entries are treated as non-edges

Returns **csgraph** : MaskedArray

masked array representation of graph

`scipy.sparse.csgraph.`**`csgraph_to_dense`**`(`*csgraph*, *null_value=0*`)`

Convert a sparse graph representation to a dense representation

New in version 0.11.0.

Parameters **csgraph** : csr_matrix, csc_matrix, or lil_matrix

Sparse representation of a graph.

null_value : float, optional

The value used to indicate null edges in the dense representation. Default is 0.

Returns **graph** : ndarray

The dense representation of the sparse graph.

Notes

For normal sparse graph representations, calling csgraph_to_dense with null_value=0 produces an equivalent result to using dense format conversions in the main sparse package. When the sparse representations have repeated values, however, the results will differ. The tools in scipy.sparse will add repeating values to obtain a final value. This function will select the minimum among repeating values to obtain a final value. For example, here we'll create a two-node directed sparse graph with multiple edges from node 0 to node 1, of weights 2 and 3. This illustrates the difference in behavior:

```
>>> from scipy.sparse import csr_matrix, csgraph
>>> data = np.array([2, 3])
>>> indices = np.array([1, 1])
>>> indptr = np.array([0, 2, 2])
>>> M = csr_matrix((data, indices, indptr), shape=(2, 2))
>>> M.toarray()
array([[0, 5],
       [0, 0]])
>>> csgraph.csgraph_to_dense(M)
array([[0., 2.],
       [0., 0.]])
```

The reason for this difference is to allow a compressed sparse graph to represent multiple edges between any two nodes. As most sparse graph algorithms are concerned with the single lowest-cost edge between any two nodes, the default scipy.sparse behavior of summming multiple weights does not make sense in this context.

The other reason for using this routine is to allow for graphs with zero-weight edges. Let's look at the example of a two-node directed graph, connected by an edge of weight zero:

```
>>> from scipy.sparse import csr_matrix, csgraph
>>> data = np.array([0.0])
>>> indices = np.array([1])
>>> indptr = np.array([0, 1, 1])
>>> M = csr_matrix((data, indices, indptr), shape=(2, 2))
>>> M.toarray()
array([[0, 0],
       [0, 0]])
>>> csgraph.csgraph_to_dense(M, np.inf)
array([[ inf,   0.],
       [ inf,  inf]])
```

In the first case, the zero-weight edge gets lost in the dense representation. In the second case, we can choose a different null value and see the true form of the graph.

`scipy.sparse.csgraph.`**`csgraph_to_masked`**`(`*csgraph*`)`

Convert a sparse graph representation to a masked array representation

New in version 0.11.0.

Parameters
: **csgraph** : csr_matrix, csc_matrix, or lil_matrix
    Sparse representation of a graph.

Returns
: **graph** : MaskedArray
    The masked dense representation of the sparse graph.

`scipy.sparse.csgraph.reconstruct_path(csgraph, predecessors, directed=True)`

Construct a tree from a graph and a predecessor list.

New in version 0.11.0.

Parameters
: **csgraph** : array_like or sparse matrix
    The N x N matrix representing the directed or undirected graph from which the predecessors are drawn.
: **predecessors** : array_like, one dimension
    The length-N array of indices of predecessors for the tree. The index of the parent of node i is given by predecessors[i].
: **directed** : bool, optional
    If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then operate on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns
: **cstree** : csr matrix
    The N x N directed compressed-sparse representation of the tree drawn from csgraph which is encoded by the predecessor list.

Graph Representations

This module uses graphs which are stored in a matrix format. A graph with N nodes can be represented by an (N x N) adjacency matrix G. If there is a connection from node i to node j, then G[i, j] = w, where w is the weight of the connection. For nodes i and j which are not connected, the value depends on the representation:

- for dense array representations, non-edges are represented by G[i, j] = 0, infinity, or NaN.
- for dense masked representations (of type np.ma.MaskedArray), non-edges are represented by masked values. This can be useful when graphs with zero-weight edges are desired.
- for sparse array representations, non-edges are represented by non-entries in the matrix. This sort of sparse representation also allows for edges with zero weights.

As a concrete example, imagine that you would like to represent the following undirected graph:

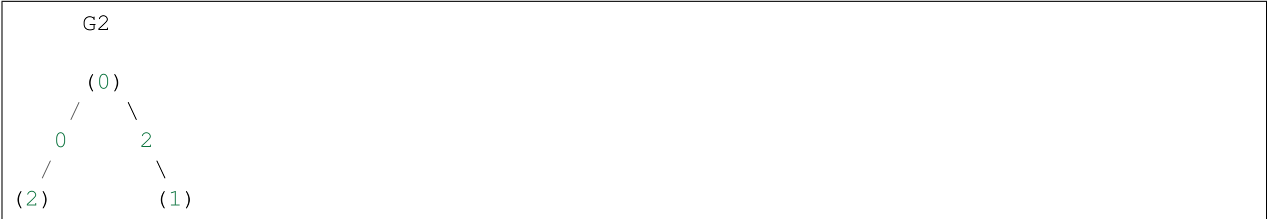
```
      G

     (0)
    /   \
   1     2
  /       \
(2)       (1)
```

This graph has three nodes, where node 0 and 1 are connected by an edge of weight 2, and nodes 0 and 2 are connected by an edge of weight 1. We can construct the dense, masked, and sparse representations as follows, keeping in mind that an undirected graph is represented by a symmetric matrix:

```
>>> G_dense = np.array([[0, 2, 1],
...                     [2, 0, 0],
...                     [1, 0, 0]])
>>> G_masked = np.ma.masked_values(G_dense, 0)
```

```
>>> from scipy.sparse import csr_matrix
>>> G_sparse = csr_matrix(G_dense)
```

This becomes more difficult when zero edges are significant. For example, consider the situation when we slightly modify the above graph:



This is identical to the previous graph, except nodes 0 and 2 are connected by an edge of zero weight. In this case, the dense representation above leads to ambiguities: how can non-edges be represented if zero is a meaningful value? In this case, either a masked or sparse representation must be used to eliminate the ambiguity:

```
>>> G2_data = np.array([[np.inf, 2,      0      ],
...                     [2,      np.inf, np.inf],
...                     [0,      np.inf, np.inf]])
>>> G2_masked = np.ma.masked_invalid(G2_data)
>>> from scipy.sparse.csgraph import csgraph_from_dense
>>> # G2_sparse = csr_matrix(G2_data) would give the wrong result
>>> G2_sparse = csgraph_from_dense(G2_data, null_value=np.inf)
>>> G2_sparse.data
array([ 2.,  0.,  2.,  0.])
```

Here we have used a utility routine from the csgraph submodule in order to convert the dense representation to a sparse representation which can be understood by the algorithms in submodule. By viewing the data array, we can see that the zero values are explicitly encoded in the graph.

Directed vs. Undirected

Matrices may represent either directed or undirected graphs. This is specified throughout the csgraph module by a boolean keyword. Graphs are assumed to be directed by default. In a directed graph, traversal from node i to node j can be accomplished over the edge G[i, j], but not the edge G[j, i]. In a non-directed graph, traversal from node i to node j can be accomplished over either G[i, j] or G[j, i]. If both edges are not null, and the two have unequal weights, then the smaller of the two is used. Note that a symmetric matrix will represent an undirected graph, regardless of whether the 'directed' keyword is set to True or False. In this case, using `directed=True` generally leads to more efficient computation.

The routines in this module accept as input either scipy.sparse representations (csr, csc, or lil format), masked representations, or dense representations with non-edges indicated by zeros, infinities, and NaN entries.

Functions

bellman_ford(csgraph[, directed, indices, ...])	Compute the shortest path lengths using the Bellman-Ford algorithm.
breadth_first_order(csgraph, i_start[, ...])	Return a breadth-first ordering starting with specified node.
breadth_first_tree(csgraph, i_start[, directed])	Return the tree generated by a breadth-first search
connected_components(csgraph[, directed, ...])	Analyze the connected components of a sparse graph
construct_dist_matrix(graph, predecessors[, ...])	Construct distance matrix from a predecessor matrix

Continued on next page

Table 5.172 – continued from previous page

cs_graph_components(*args, **kwds)	cs_graph_components is deprecated!
csgraph_from_dense(graph[, null_value, ...])	Construct a CSR-format sparse graph from a dense matrix.
csgraph_from_masked(graph)	Construct a CSR-format graph from a masked array.
csgraph_masked_from_dense(graph[, ...])	Construct a masked array graph representation from a dense matrix.
csgraph_to_dense(csgraph[, null_value])	Convert a sparse graph representation to a dense representation
csgraph_to_masked(csgraph)	Convert a sparse graph representation to a masked array representation
depth_first_order(csgraph, i_start[, ...])	Return a depth-first ordering starting with specified node.
depth_first_tree(csgraph, i_start[, directed])	Return a tree generated by a depth-first search.
dijkstra(csgraph[, directed, indices, ...])	Dijkstra algorithm using Fibonacci Heaps
floyd_warshall(csgraph[, directed, ...])	Compute the shortest path lengths using the Floyd-Warshall algorithm
johnson(csgraph[, directed, indices, ...])	Compute the shortest path lengths using Johnson's algorithm.
laplacian(csgraph[, normed, return_diag, ...])	Return the Laplacian matrix of a directed graph.
maximum_bipartite_matching(graph[, perm_type])	Returns an array of row or column permutations that makes the diagonal of a nonsingular square CSC sparse matrix zero free.
minimum_spanning_tree(csgraph[, overwrite])	Return a minimum spanning tree of an undirected graph
reconstruct_path(csgraph, predecessors[, ...])	Construct a tree from a graph and a predecessor list.
reverse_cuthill_mckee(graph[, symmetric_mode])	Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering.
shortest_path(csgraph[, method, directed, ...])	Perform a shortest-path graph search on a positive directed or undirected graph.
structural_rank(graph)	Compute the structural rank of a graph (matrix) with a given sparsity pattern.

Classes

Tester	alias of NoseTester

Exceptions

NegativeCycleError

Sparse linear algebra (scipy.sparse.linalg)

Abstract linear operators

LinearOperator(dtype, shape)	Common interface for performing matrix vector products
aslinearoperator(A)	Return A as a LinearOperator.

class scipy.sparse.linalg.**LinearOperator**(*dtype*, *shape*)

Common interface for performing matrix vector products

Many iterative methods (e.g. cg, gmres) do not need to know the individual entries of a matrix to solve a linear

system A*x=b. Such solvers only require the computation of matrix vector products, A*v where v is a dense vector. This class serves as an abstract interface between iterative solvers and matrix-like objects.

To construct a concrete LinearOperator, either pass appropriate callables to the constructor of this class, or subclass it.

A subclass must implement either one of the methods `_matvec` and `_matmat`, and the attributes/properties `shape` (pair of integers) and `dtype` (may be None). It may call the `__init__` on this class to have these attributes validated. Implementing `_matvec` automatically implements `_matmat` (using a naive algorithm) and vice-versa.

Optionally, a subclass may implement `_rmatvec` or `_adjoint` to implement the Hermitian adjoint (conjugate transpose). As with `_matvec` and `_matmat`, implementing either `_rmatvec` or `_adjoint` implements the other automatically. Implementing `_adjoint` is preferable; `_rmatvec` is mostly there for backwards compatibility.

Parameters

shape : tuple
: Matrix dimensions (M,N).

matvec : callable f(v)
: Returns returns A * v.

rmatvec : callable f(v)
: Returns A^H * v, where A^H is the conjugate transpose of A.

matmat : callable f(V)
: Returns A * V, where V is a dense matrix with dimensions (N,K).

dtype : dtype
: Data type of the matrix.

See also:

`aslinearoperator`
Construct LinearOperators

Notes

The user-defined matvec() function must properly handle the case where v has shape (N,) as well as the (N,1) case. The shape of the return type is handled internally by LinearOperator.

LinearOperator instances can also be multiplied, added with each other and exponentiated, all lazily: the result of these operations is always a new, composite LinearOperator, that defers linear operations to the original operators and combines the results.

Examples

```
>>> import numpy as np
>>> from scipy.sparse.linalg import LinearOperator
>>> def mv(v):
...     return np.array([2*v[0], 3*v[1]])
...
>>> A = LinearOperator((2,2), matvec=mv)
>>> A
<2x2 _CustomLinearOperator with dtype=float64>
>>> A.matvec(np.ones(2))
array([ 2.,  3.])
>>> A * np.ones(2)
array([ 2.,  3.])
```

Attributes

args	(tuple) For linear operators describing products etc. of other linear operators, the operands of the binary operation.

Methods

`__call__`(x)	
`adjoint`()	Hermitian adjoint.
`dot`(x)	Matrix-matrix or matrix-vector multiplication.
`matmat`(X)	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`transpose`()	Transpose this linear operator.

`LinearOperator.`**`__call__`**(*x*)

`LinearOperator.`**`adjoint`**()

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** : LinearOperator

Hermitian adjoint of self.

`LinearOperator.`**`dot`**(*x*)

Matrix-matrix or matrix-vector multiplication.

Parameters **x** : array_like

1-d or 2-d array, representing a vector or matrix.

Returns **Ax** : array

1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

`LinearOperator.`**`matmat`**(*X*)

Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** : {matrix, ndarray}

An array with shape (N,K).

Returns **Y** : {matrix, ndarray}

A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

`LinearOperator.`**`matvec`**(*x*)

Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}

An array with shape (N,) or (N,1).

Returns **y** : {matrix, ndarray}
A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

`LinearOperator.rmatvec(x)`
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}
An array with shape (M,) or (M,1).

Returns **y** : {matrix, ndarray}
A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

`LinearOperator.transpose()`
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

`scipy.sparse.linalg.aslinearoperator(A)`
Return A as a LinearOperator.
'A' may be any of the following types:

- ndarray
- matrix
- sparse matrix (e.g. csr_matrix, lil_matrix, etc.)
- LinearOperator
- An object with .shape and .matvec attributes

See the LinearOperator documentation for additional information.

Examples

```
>>> from scipy.sparse.linalg import aslinearoperator
>>> M = np.array([[1,2,3],[4,5,6]], dtype=np.int32)
>>> aslinearoperator(M)
<2x3 MatrixLinearOperator with dtype=int32>
```

Matrix Operations

inv(A)	Compute the inverse of a sparse matrix
expm(A)	Compute the matrix exponential using Pade approximation.
expm_multiply(A, B[, start, stop, num, endpoint])	Compute the action of the matrix exponential of A on B.

`scipy.sparse.linalg.`**`inv`**`(A)`

Compute the inverse of a sparse matrix

Parameters **A** : (M,M) ndarray or sparse matrix
square matrix to be inverted

Returns **Ainv** : (M,M) ndarray or sparse matrix
inverse of *A*

Notes

This computes the sparse inverse of *A*. If the inverse of *A* is expected to be non-sparse, it will likely be faster to convert *A* to dense and use scipy.linalg.inv.

New in version 0.12.0.

`scipy.sparse.linalg.`**`expm`**`(A)`

Compute the matrix exponential using Pade approximation.

Parameters **A** : (M,M) array_like or sparse matrix
2D Array or Matrix (sparse or dense) to be exponentiated

Returns **expA** : (M,M) ndarray
Matrix exponential of *A*

Notes

This is algorithm (6.1) which is a simplification of algorithm (5.1).

New in version 0.12.0.

References

[R21]

`scipy.sparse.linalg.`**`expm_multiply`**`(`*A*, *B*, *start=None*, *stop=None*, *num=None*, *endpoint=None*`)`

Compute the action of the matrix exponential of A on B.

Parameters **A** : transposable linear operator
The operator whose exponential is of interest.

B : ndarray
The matrix or vector to be multiplied by the matrix exponential of A.

start : scalar, optional
The starting time point of the sequence.

stop : scalar, optional
The end time point of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced time points, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : int, optional
Number of time points to use.

endpoint : bool, optional
If True, *stop* is the last time point. Otherwise, it is not included.

Returns **expm_A_B** : ndarray
The result of the action $e^{t_k A} B$.

Notes

The optional arguments defining the sequence of evenly spaced time points are compatible with the arguments of `numpy.linspace`.

The output ndarray shape is somewhat complicated so I explain it here. The ndim of the output could be either 1, 2, or 3. It would be 1 if you are computing the expm action on a single vector at a single time point. It would be 2 if you are computing the expm action on a vector at multiple time points, or if you are computing the expm

action on a matrix at a single time point. It would be 3 if you want the action on a matrix with multiple columns at multiple time points. If multiple time points are requested, expm_A_B[0] will always be the action of the expm at the first time point, regardless of whether the action is on a vector or a matrix.

References

[R22], *[R23]*

Matrix norms

norm(x[, ord, axis])	Norm of a sparse matrix
onenormest(A[, t, itmax, compute_v, compute_w])	Compute a lower bound of the 1-norm of a sparse matrix.

`scipy.sparse.linalg.`**`norm`**(*x*, *ord=None*, *axis=None*)

Norm of a sparse matrix

This function is able to return one of seven different matrix norms, depending on the value of the `ord` parameter.

Parameters

x : a sparse matrix
Input sparse matrix.

ord : {non-zero int, inf, -inf, 'fro'}, optional
Order of the norm (see table under `Notes`). inf means numpy's *inf* object.

axis : {int, 2-tuple of ints, None}, optional
If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

Returns

n : float or ndarray

Notes

Some of the ord are not implemented because some associated functions like, _multi_svd_norm, are not yet available for sparse matrix.

This docstring is modified based on numpy.linalg.norm. https://github.com/numpy/numpy/blob/master/numpy/linalg/linalg.py

The following norms can be calculated:

ord	norm for sparse matrices
None	Frobenius norm
'fro'	Frobenius norm
inf	max(sum(abs(x), axis=1))
-inf	min(sum(abs(x), axis=1))
0	abs(x).sum(axis=axis)
1	max(sum(abs(x), axis=0))
-1	min(sum(abs(x), axis=0))
2	Not implemented
-2	Not implemented
other	Not implemented

The Frobenius norm is given by *[R34]*:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

References

[R34]

Examples

```
>>> from scipy.sparse import *
>>> import numpy as np
>>> from scipy.sparse.linalg import norm
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
>>> b = a.reshape((3, 3))
>>> b
array([[-4, -3, -2],
       [-1, 0, 1],
       [ 2, 3, 4]])
```

```
>>> b = csr_matrix(b)
>>> norm(b)
7.745966692414834
>>> norm(b, 'fro')
7.745966692414834
>>> norm(b, np.inf)
9
>>> norm(b, -np.inf)
2
>>> norm(b, 1)
7
>>> norm(b, -1)
6
```

`scipy.sparse.linalg.`**`onenormest`**(*A*, *t=2*, *itmax=5*, *compute_v=False*, *compute_w=False*)

Compute a lower bound of the 1-norm of a sparse matrix.

Parameters

A : ndarray or other linear operator
: A linear operator that can be transposed and that can produce matrix products.

t : int, optional
: A positive parameter controlling the tradeoff between accuracy versus time and memory usage. Larger values take longer and use more memory but give more accurate output.

itmax : int, optional
: Use at most this many iterations.

compute_v : bool, optional
: Request a norm-maximizing linear operator input vector if True.

compute_w : bool, optional
: Request a norm-maximizing linear operator output vector if True.

Returns

est : float
: An underestimate of the 1-norm of the sparse matrix.

v : ndarray, optional
: The vector such that ||Av||_1 == est*||v||_1. It can be thought of as an input to the linear operator that gives an output with particularly large norm.

w : ndarray, optional
: The vector Av which has relatively large 1-norm. It can be thought of as an output of the linear operator that is relatively large in norm compared to the input.

Notes

This is algorithm 2.4 of [1].

In [2] it is described as follows. “This algorithm typically requires the evaluation of about 4t matrix-vector products and almost invariably produces a norm estimate (which is, in fact, a lower bound on the norm) correct to within a factor 3.”

New in version 0.13.0.

References

[R35], [R36]

Solving linear problems

Direct methods for linear equation systems:

spsolve(A, b[, permc_spec, use_umfpack])	Solve the sparse linear system Ax=b, where b may be a vector or a matrix.
spsolve_triangular(A, b[, lower, ...])	Solve the equation $A x = b$ for x, assuming A is a triangular matrix.
factorized(A)	Return a function for solving a sparse linear system, with A pre-factorized.
MatrixRankWarning	
use_solver(**kwargs)	Select default sparse direct solver to be used.

scipy.sparse.linalg.**spsolve**(*A*, *b*, *permc_spec=None*, *use_umfpack=True*)
Solve the sparse linear system Ax=b, where b may be a vector or a matrix.

Parameters

A : ndarray or sparse matrix
The square matrix A will be converted into CSC or CSR form

b : ndarray or sparse matrix
The matrix or vector representing the right hand side of the equation. If a vector, b.shape must be (n,) or (n, 1).

permc_spec : str, optional
How to permute the columns of the matrix for sparsity preservation. (default: ‘COLAMD’)

- `NATURAL`: natural ordering.
- `MMD_ATA`: minimum degree ordering on the structure of A^T A.
- `MMD_AT_PLUS_A`: minimum degree ordering on the structure of A^T+A.
- `COLAMD`: approximate minimum degree column ordering

use_umfpack : bool, optional
if True (default) then use umfpack for the solution. This is only referenced if b is a vector and `scikit-umfpack` is installed.

Returns

x : ndarray or sparse matrix
the solution of the sparse linear equation. If b is a vector, then x is a vector of size A.shape[1] If b is a matrix, then x is a matrix of size (A.shape[1], b.shape[1])

Notes

For solving the matrix expression AX = B, this solver assumes the resulting matrix X is sparse, as is often the case for very sparse inputs. If the resulting X is dense, the construction of this sparse result will be relatively expensive. In that case, consider converting A to a dense matrix and using scipy.linalg.solve or its variants.

`scipy.sparse.linalg.`**`spsolve_triangular`**(*A*, *b*, *lower=True*, *overwrite_A=False*, *overwrite_b=False*)

Solve the equation $A x = b$ for x, assuming A is a triangular matrix.

Parameters

A : (M, M) sparse matrix
: A sparse square triangular matrix. Should be in CSR format.

b : (M,) or (M, N) array_like
: Right-hand side matrix in $A x = b$

lower : bool, optional
: Whether *A* is a lower or upper triangular matrix. Default is lower triangular matrix.

overwrite_A : bool, optional
: Allow changing *A*. The indices of *A* are going to be sorted and zero entries are going to be removed. Enabling gives a performance gain. Default is False.

overwrite_b : bool, optional
: Allow overwriting data in *b*. Enabling gives a performance gain. Default is False.

Returns

x : (M,) or (M, N) ndarray
: Solution to the system $A x = b$. Shape of return matches shape of *b*.

Raises

LinAlgError
: If *A* is singular or not triangular.

ValueError
: If shape of *A* or shape of *b* do not match the requirements.

Notes

New in version 0.19.0.

`scipy.sparse.linalg.`**`factorized`**(*A*)

Return a function for solving a sparse linear system, with A pre-factorized.

Parameters

A : (N, N) array_like
: Input.

Returns

solve : callable
: To solve the linear system of equations given in *A*, the *solve* callable should be passed an ndarray of shape (N,).

Examples

```
>>> from scipy.sparse.linalg import factorized
>>> A = np.array([[ 3. ,  2. , -1. ],
...               [ 2. , -2. ,  4. ],
...               [-1. ,  0.5, -1. ]])
>>> solve = factorized(A) # Makes LU decomposition.
>>> rhs1 = np.array([1, -2, 0])
>>> solve(rhs1) # Uses the LU factors.
array([ 1., -2., -2.])
```

exception `scipy.sparse.linalg.`**`MatrixRankWarning`**

`scipy.sparse.linalg.`**`use_solver`**(***kwargs*)

Select default sparse direct solver to be used.

Parameters

useUmfpack : bool, optional
: Use UMFPACK over SuperLU. Has effect only if scikits.umfpack is installed. Default: True

Notes

The default sparse solver is umfpack when available (scikits.umfpack is installed). This can be changed by passing useUmfpack = False, which then causes the always present SuperLU based solver to be used.

Umfpack requires a CSR/CSC matrix to have sorted column/row indices. If sure that the matrix fulfills this, pass `assumeSortedIndices=True` to gain some speed.

Iterative methods for linear equation systems:

bicg(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient iteration to solve `Ax = b`.
bicgstab(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient STABilized iteration to solve `Ax = b`.
cg(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient iteration to solve `Ax = b`.
cgs(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient Squared iteration to solve `Ax = b`.
gmres(A, b[, x0, tol, restart, maxiter, ...])	Use Generalized Minimal RESidual iteration to solve `Ax = b`.
lgmres(A, b[, x0, tol, maxiter, M, ...])	Solve a matrix equation using the LGMRES algorithm.
minres(A, b[, x0, shift, tol, maxiter, ...])	Use MINimum RESidual iteration to solve Ax=b
qmr(A, b[, x0, tol, maxiter, xtype, M1, M2, ...])	Use Quasi-Minimal Residual iteration to solve `Ax = b`.

`scipy.sparse.linalg.`**`bicg`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use BIConjugate Gradient iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real or complex N-by-N matrix of the linear system. It is required that the linear operator can produce `Ax` and `A^T x`.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {‘f’,’d’,’F’,’D’}

This parameter is deprecated – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`bicgstab`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use BIConjugate Gradient STABilized iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
The real or complex N-by-N matrix of the linear system.

b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}
The converged solution.

info : integer
Provides convergence information:
0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}
Starting guess for the solution.

tol : float
Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer
Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}
Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
This parameter is deprecated – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`cg`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use Conjugate Gradient iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
The real or complex N-by-N matrix of the linear system. `A` must represent a hermitian, positive definite matrix.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns **x** : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}

This parameter is deprecated – avoid using it.

The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`cgs`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use Conjugate Gradient Squared iteration to solve `Ax = b`.

Parameters **A** : {sparse matrix, dense matrix, LinearOperator}

The real-valued N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns **x** : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}

This parameter is deprecated – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`gmres`**(*A*, *b*, *x0=None*, *tol=1e-05*, *restart=None*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*, *restrt=None*)

Use Generalized Minimal RESidual iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real or complex N-by-N matrix of the linear system.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : int

Provides convergence information:

- 0 : successful exit
- >0 : convergence to tolerance not achieved, number of iterations
- <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution (a vector of zeros by default).

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

restart : int, optional

Number of iterations between restarts. Larger values increase iteration cost, but may be necessary for convergence. Default is 20.

maxiter : int, optional

Maximum number of iterations (restart cycles). Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

xtype : {'f','d','F','D'}

This parameter is DEPRECATED — avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

M : {sparse matrix, dense matrix, LinearOperator}
Inverse of the preconditioner of A. M should approximate the inverse of A and be easy to solve for (see Notes). Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance. By default, no preconditioner is used.

callback : function
User-supplied function to call after each iteration. It is called as callback(rk), where rk is the current residual vector.

restrt : int, optional
DEPRECATED - use *restart* instead.

See also:

`LinearOperator`

Notes

A preconditioner, P, is chosen such that P is close to A but easy to solve for. The preconditioner parameter required by this routine is `M = P^-1`. The inverse should preferably not be calculated explicitly. Rather, use the following template to produce M:

```
# Construct a linear operator that computes P^-1 * x.
import scipy.sparse.linalg as spla
M_x = lambda x: spla.spsolve(P, x)
M = spla.LinearOperator((n, n), M_x)
```

`scipy.sparse.linalg.`**`lgmres`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=1000*, *M=None*, *callback=None*, *inner_m=30*, *outer_k=3*, *outer_v=None*, *store_outer_Av=True*)

Solve a matrix equation using the LGMRES algorithm.

The LGMRES algorithm *[R24] [R25]* is designed to avoid some problems in the convergence in restarted GMRES, and often converges in fewer iterations.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
The real or complex N-by-N matrix of the linear system.

b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

x0 : {array, matrix}
Starting guess for the solution.

tol : float, optional
Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : int, optional
Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}, optional
Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function, optional
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

inner_m : int, optional
Number of inner GMRES iterations per each outer iteration.

outer_k : int, optional

Number of vectors to carry between inner GMRES iterations. According to *[R24]*, good values are in the range of 1...3. However, note that if you want to use the additional vectors to accelerate solving multiple similar problems, larger values may be beneficial.

outer_v : list of tuples, optional

List containing tuples `(v, Av)` of vectors and corresponding matrix-vector products, used to augment the Krylov subspace, and carried between inner GMRES iterations. The element `Av` can be *None* if the matrix-vector product should be re-evaluated. This parameter is modified in-place by `lgmres`, and can be used to pass "guess" vectors in and out of the algorithm when solving similar problems.

store_outer_Av : bool, optional

Whether LGMRES should store also A*v in addition to vectors *v* in the *outer_v* list. Default is True.

Returns

x : array or matrix

The converged solution.

info : int

Provides convergence information:

- 0 : successful exit
- >0 : convergence to tolerance not achieved, number of iterations
- <0 : illegal input or breakdown

Notes

The LGMRES algorithm *[R24] [R25]* is designed to avoid the slowing of convergence in restarted GMRES, due to alternating residual vectors. Typically, it often outperforms GMRES(m) of comparable memory requirements by some measure, or at least is not much worse.

Another advantage in this algorithm is that you can supply it with 'guess' vectors in the *outer_v* argument that augment the Krylov subspace. If the solution lies close to the span of these vectors, the algorithm converges faster. This can be useful if several very similar matrices need to be inverted one after another, such as in Newton-Krylov iteration where the Jacobian matrix often changes little in the nonlinear steps.

References

[R24], *[R25]*

`scipy.sparse.linalg.`**`minres`**(*A*, *b*, *x0=None*, *shift=0.0*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*, *show=False*, *check=False*)

Use MINimum RESidual iteration to solve Ax=b

MINRES minimizes norm(A*x - b) for a real symmetric matrix A. Unlike the Conjugate Gradient method, A can be indefinite or singular.

If shift != 0 then the method solves (A - shift*I)x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real symmetric N-by-N matrix of the linear system

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}

This parameter is deprecated – avoid using it.

The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

Notes

THIS FUNCTION IS EXPERIMENTAL AND SUBJECT TO CHANGE!

References

Solution of sparse indefinite systems of linear equations,

C. C. Paige and M. A. Saunders (1975), SIAM J. Numer. Anal. 12(4), pp. 617-629. http://www.stanford.edu/group/SOL/software/minres.html

This file is a translation of the following MATLAB implementation:

http://www.stanford.edu/group/SOL/software/minres/matlab/

`scipy.sparse.linalg.`**`qmr`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M1=None*, *M2=None*, *callback=None*)

Use Quasi-Minimal Residual iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real-valued N-by-N matrix of the linear system. It is required that the linear operator can produce `Ax` and `A^T x`.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer
Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M1 : {sparse matrix, dense matrix, LinearOperator}
Left preconditioner for A.

M2 : {sparse matrix, dense matrix, LinearOperator}
Right preconditioner for A. Used together with the left preconditioner M1. The matrix M1*A*M2 should have better conditioned than A alone.

callback : function
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
This parameter is DEPRECATED – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

See also:

`LinearOperator`

Iterative methods for least-squares problems:

`lsqr`(A, b[, damp, atol, btol, conlim, ...])	Find the least-squares solution to a large, sparse, linear system of equations.
`lsmr`(A, b[, damp, atol, btol, conlim, ...])	Iterative solver for least-squares problems.

`scipy.sparse.linalg.`**`lsqr`**(*A, b, damp=0.0, atol=1e-08, btol=1e-08, conlim=100000000.0, iter_lim=None, show=False, calc_var=False*)

Find the least-squares solution to a large, sparse, linear system of equations.

The function solves `Ax = b` or `min ||b - Ax||^2` or `min ||Ax - b||^2 + d^2 ||x||^2`.

The matrix A may be square or rectangular (over-determined or under-determined), and may have any rank.

```
1. Unsymmetric equations --    solve  A*x = b

2. Linear least squares  --    solve  A*x = b
                               in the least-squares sense

3. Damped least squares  --    solve  (   A    )*x = ( b )
                                      ( damp*I )     ( 0 )
                               in the least-squares sense
```

Parameters **A** : {sparse matrix, ndarray, LinearOperator}
Representation of an m-by-n matrix. It is required that the linear operator can produce `Ax` and `A^T x`.

b : array_like, shape (m,)
Right-hand side vector `b`.

damp : float
Damping coefficient.

atol, btol : float, optional
Stopping tolerances. If both are 1.0e-9 (say), the final residual norm should be accurate to about 9 digits. (The final x will usually have fewer correct digits, depending on cond(A) and the size of damp.)

conlim : float, optional
Another stopping tolerance. lsqr terminates if an estimate of `cond(A)` exceeds *conlim*. For compatible systems `Ax = b`, *conlim* could be as large as 1.0e+12 (say). For least-squares problems, conlim should be less than 1.0e+8. Maximum precision can be obtained by setting `atol = btol = conlim = zero`, but the number of iterations may then be excessive.

iter_lim : int, optional
Explicit limitation on number of iterations (for safety).

show : bool, optional
Display an iteration log.

calc_var : bool, optional
Whether to estimate diagonals of `(A'A + damp^2*I)^{-1}`.

Returns

x : ndarray of float
The final solution.

istop : int
Gives the reason for termination. 1 means x is an approximate solution to Ax = b. 2 means x approximately solves the least-squares problem.

itn : int
Iteration number upon termination.

r1norm : float
`norm(r)`, where `r = b - Ax`.

r2norm : float
`sqrt( norm(r)^2 + damp^2 * norm(x)^2 )`. Equal to *r1norm* if `damp == 0`.

anorm : float
Estimate of Frobenius norm of `Abar = [[A]; [damp*I]]`.

acond : float
Estimate of `cond(Abar)`.

arnorm : float
Estimate of `norm(A'*r - damp^2*x)`.

xnorm : float
`norm(x)`

var : ndarray of float
If `calc_var` is True, estimates all diagonals of `(A'A)^{-1}` (if `damp == 0`) or more generally `(A'A + damp^2*I)^{-1}`. This is well defined if A has full column rank or `damp > 0`. (Not sure what var means if `rank(A) < n` and `damp = 0.`)

Notes

LSQR uses an iterative method to approximate the solution. The number of iterations required to reach a certain accuracy depends strongly on the scaling of the problem. Poor scaling of the rows or columns of A should therefore be avoided where possible.

For example, in problem 1 the solution is unaltered by row-scaling. If a row of A is very small or large compared to the other rows of A, the corresponding row of (A b) should be scaled up or down.

In problems 1 and 2, the solution x is easily recovered following column-scaling. Unless better information is known, the nonzero columns of A should be scaled so that they all have the same Euclidean norm (e.g., 1.0).

In problem 3, there is no freedom to re-scale if damp is nonzero. However, the value of damp should be assigned only after attention has been paid to the scaling of A.

The parameter damp is intended to help regularize ill-conditioned systems, by preventing the true solution from being very large. Another aid to regularization is provided by the parameter acond, which may be used to terminate iterations before the computed solution becomes very large.

If some initial estimate `x0` is known and if `damp == 0`, one could proceed as follows:

1. Compute a residual vector `r0 = b - A*x0`.
2. Use LSQR to solve the system `A*dx = r0`.
3. Add the correction dx to obtain a final solution `x = x0 + dx`.

This requires that `x0` be available before and after the call to LSQR. To judge the benefits, suppose LSQR takes k1 iterations to solve A*x = b and k2 iterations to solve A*dx = r0. If x0 is "good", norm(r0) will be smaller than norm(b). If the same stopping tolerances atol and btol are used for each system, k1 and k2 will be similar, but the final solution x0 + dx should be more accurate. The only way to reduce the total work is to use a larger stopping tolerance for the second system. If some value btol is suitable for A*x = b, the larger value btol*norm(b)/norm(r0) should be suitable for A*dx = r0.

Preconditioning is another way to reduce the number of iterations. If it is possible to solve a related system `M*x = b` efficiently, where M approximates A in some helpful way (e.g. M - A has low rank or its elements are small relative to those of A), LSQR may converge more rapidly on the system `A*M(inverse)*z = b`, after which x can be recovered by solving M*x = z.

If A is symmetric, LSQR should not be used!

Alternatives are the symmetric conjugate-gradient method (cg) and/or SYMMLQ. SYMMLQ is an implementation of symmetric cg that applies to any symmetric A and will converge more rapidly than LSQR. If A is positive definite, there are other implementations of symmetric cg that require slightly less work per iteration than SYMMLQ (but will take the same number of iterations).

References

[R31], *[R32]*, *[R33]*

`scipy.sparse.linalg.`**`lsmr`**(*A*, *b*, *damp=0.0*, *atol=1e-06*, *btol=1e-06*, *conlim=100000000.0*, *maxiter=None*, *show=False*)

Iterative solver for least-squares problems.

lsmr solves the system of linear equations `Ax = b`. If the system is inconsistent, it solves the least-squares problem `min ||b - Ax||_2`. A is a rectangular matrix of dimension m-by-n, where all cases are allowed: m = n, m > n, or m < n. B is a vector of length m. The matrix A may be dense or sparse (usually sparse).

Parameters

A : {matrix, sparse matrix, ndarray, LinearOperator}

Matrix A in the linear system.

b : array_like, shape (m,)

Vector b in the linear system.

damp : float

Damping factor for regularized least-squares. *lsmr* solves the regularized least-squares problem:

```
min ||(b) - (   A   )x||
    ||(0)    (damp*I) ||_2
```

where damp is a scalar. If damp is None or 0, the system is solved without regularization.

atol, btol : float, optional

Stopping tolerances. *lsmr* continues iterations until a certain backward error estimate is smaller than some quantity depending on atol and btol. Let `r = b - Ax` be the residual vector for the current approximate solution x. If `Ax = b` seems to be consistent, `lsmr` terminates when `norm(r) <= atol * norm(A) * norm(x) + btol * norm(b)`. Otherwise, lsmr terminates when `norm(A^{T} r) <= atol * norm(A)`

`* norm(r)`. If both tolerances are 1.0e-6 (say), the final `norm(r)` should be accurate to about 6 digits. (The final x will usually have fewer correct digits, depending on `cond(A)` and the size of LAMBDA.) If *atol* or *btol* is None, a default value of 1.0e-6 will be used. Ideally, they should be estimates of the relative error in the entries of A and B respectively. For example, if the entries of *A* have 7 correct digits, set atol = 1e-7. This prevents the algorithm from doing unnecessary work beyond the uncertainty of the input data.

conlim : float, optional

`lsmr` terminates if an estimate of `cond(A)` exceeds *conlim*. For compatible systems `Ax = b`, conlim could be as large as 1.0e+12 (say). For least-squares problems, *conlim* should be less than 1.0e+8. If *conlim* is None, the default value is 1e+8. Maximum precision can be obtained by setting `atol = btol = conlim = 0`, but the number of iterations may then be excessive.

maxiter : int, optional

`lsmr` terminates if the number of iterations reaches *maxiter*. The default is `maxiter = min(m, n)`. For ill-conditioned systems, a larger value of *maxiter* may be needed.

show : bool, optional

Print iterations logs if `show=True`.

Returns **x** : ndarray of float

Least-square solution returned.

istop : int

istop gives the reason for stopping:

```
istop   = 0 means x=0 is a solution.
        = 1 means x is an approximate solution to A*x␣
↪= B,
          according to atol and btol.
        = 2 means x approximately solves the least-
↪squares problem
          according to atol.
        = 3 means COND(A) seems to be greater than␣
↪CONLIM.
        = 4 is the same as 1 with atol = btol = eps␣
↪(machine
          precision)
        = 5 is the same as 2 with atol = eps.
        = 6 is the same as 3 with CONLIM = 1/eps.
        = 7 means ITN reached maxiter before the other␣
↪stopping
          conditions were satisfied.
```

itn : int

Number of iterations used.

normr : float

`norm(b-Ax)`

normar : float

`norm(A^T (b - Ax))`

norma : float

`norm(A)`

conda : float

Condition number of A.

normx : float

`norm(x)`

Notes

New in version 0.11.0.

References

[R29], *[R30]*

Matrix factorizations

Eigenvalue problems:

`eigs`(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the square matrix A.
`eigsh`(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex hermitian matrix A.
`lobpcg`(A, X[, B, M, Y, tol, maxiter, ...])	Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

`scipy.sparse.linalg.`**`eigs`**(*A*, *k=6*, *M=None*, *sigma=None*, *which='LM'*, *v0=None*, *ncv=None*, *maxiter=None*, *tol=0*, *return_eigenvectors=True*, *Minv=None*, *OPinv=None*, *OPpart=None*)

Find k eigenvalues and eigenvectors of the square matrix A.

Solves `A * x[i] = w[i] * x[i]`, the standard eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i].

If M is specified, solves `A * x[i] = w[i] * M * x[i]`, the generalized eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i]

Parameters

A : ndarray, sparse matrix or LinearOperator

An array, sparse matrix, or LinearOperator representing the operation `A * x`, where A is a real or complex square matrix.

k : int, optional

The number of eigenvalues and eigenvectors desired. *k* must be smaller than N-1. It is not possible to compute all eigenvectors of a matrix.

M : ndarray, sparse matrix or LinearOperator, optional

An array, sparse matrix, or LinearOperator representing the operation M*x for the generalized eigenvalue problem

A * x = w * M * x.

M must represent a real, symmetric matrix if A is real, and must represent a complex, hermitian matrix if A is complex. For best results, the data type of M should be the same as that of A. Additionally:

If *sigma* is None, M is positive definite

If sigma is specified, M is positive semi-definite

If sigma is None, eigs requires an operator to compute the solution of the linear equation `M * x = b`. This is done internally via a (sparse) LU decomposition for an explicit matrix M, or via an iterative solver for a general linear operator. Alternatively, the user can supply the matrix or operator Minv, which gives `x = Minv * b = M^-1 * b`.

sigma : real or complex, optional

Find eigenvalues near sigma using shift-invert mode. This requires an operator to compute the solution of the linear system `[A - sigma * M] * x = b`, where M is the identity matrix if unspecified. This is computed internally via a (sparse) LU decomposition for explicit matrices A & M, or

via an iterative solver if either A or M is a general linear operator. Alternatively, the user can supply the matrix or operator OPinv, which gives `x = OPinv * b = [A - sigma * M]^-1 * b`. For a real matrix A, shift-invert can either be done in imaginary mode or real mode, specified by the parameter OPpart ('r' or 'i'). Note that when sigma is specified, the keyword 'which' (below) refers to the shifted eigenvalues `w'[i]` where:

If A is real and OPpart == 'r' (default),
`w'[i] = 1/2 * [1/(w[i]-sigma) + 1/(w[i]-conj(sigma))].`

If A is real and OPpart == 'i',
`w'[i] = 1/2i * [1/(w[i]-sigma) - 1/(w[i]-conj(sigma))].`

If A is complex, `w'[i] = 1/(w[i]-sigma).`

v0 : ndarray, optional
Starting vector for iteration. Default: random

ncv : int, optional
The number of Lanczos vectors generated *ncv* must be greater than *k*; it is recommended that `ncv > 2*k`. Default: `min(n, max(2*k + 1, 20))`

which : str, ['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'], optional
Which *k* eigenvectors and eigenvalues to find:

'LM' : largest magnitude
'SM' : smallest magnitude
'LR' : largest real part
'SR' : smallest real part
'LI' : largest imaginary part
'SI' : smallest imaginary part

When sigma != None, 'which' refers to the shifted eigenvalues w'[i] (see discussion in 'sigma', above). ARPACK is generally better at finding large values than small values. If small eigenvalues are desired, consider using shift-invert mode for better performance.

maxiter : int, optional
Maximum number of Arnoldi update iterations allowed Default: `n*10`

tol : float, optional
Relative accuracy for eigenvalues (stopping criterion) The default value of 0 implies machine precision.

return_eigenvectors : bool, optional
Return eigenvectors (True) in addition to eigenvalues

Minv : ndarray, sparse matrix or LinearOperator, optional
See notes in M, above.

OPinv : ndarray, sparse matrix or LinearOperator, optional
See notes in sigma, above.

OPpart : {'r' or 'i'}, optional
See notes in sigma, above

Returns

w : ndarray
Array of k eigenvalues.

v : ndarray
An array of *k* eigenvectors. `v[:, i]` is the eigenvector corresponding to the eigenvalue w[i].

Raises

ArpackNoConvergence
When the requested convergence is not obtained. The currently converged eigenvalues and eigenvectors can be found as `eigenvalues` and `eigenvectors` attributes of the exception object.

See also:

eigsh eigenvalues and eigenvectors for symmetric matrix A
svds singular value decomposition for a matrix A

Notes

This function is a wrapper to the ARPACK *[R17]* SNEUPD, DNEUPD, CNEUPD, ZNEUPD, functions which use the Implicitly Restarted Arnoldi Method to find the eigenvalues and eigenvectors *[R18]*.

References

[R17], *[R18]*

Examples

Find 6 eigenvectors of the identity matrix:

```
>>> import scipy.sparse as sparse
>>> id = np.eye(13)
>>> vals, vecs = sparse.linalg.eigs(id, k=6)
>>> vals
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
>>> vecs.shape
(13, 6)
```

`scipy.sparse.linalg.`**`eigsh`**(*A*, *k=6*, *M=None*, *sigma=None*, *which='LM'*, *v0=None*, *ncv=None*, *maxiter=None*, *tol=0*, *return_eigenvectors=True*, *Minv=None*, *OPinv=None*, *mode='normal'*)

Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex hermitian matrix A.

Solves `A * x[i] = w[i] * x[i]`, the standard eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i].

If M is specified, solves `A * x[i] = w[i] * M * x[i]`, the generalized eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i]

Parameters

A : An N x N matrix, array, sparse matrix, or LinearOperator representing
the operation A * x, where A is a real symmetric matrix For buckling mode (see below) A must additionally be positive-definite

k : int, optional
The number of eigenvalues and eigenvectors desired. *k* must be smaller than N. It is not possible to compute all eigenvectors of a matrix.

Returns

w : array
Array of k eigenvalues

v : array
An array representing the *k* eigenvectors. The column `v[:, i]` is the eigenvector corresponding to the eigenvalue `w[i]`.

Other Parameters

M : An N x N matrix, array, sparse matrix, or linear operator representing
the operation M * x for the generalized eigenvalue problem

A * x = w * M * x.

M must represent a real, symmetric matrix if A is real, and must represent a complex, hermitian matrix if A is complex. For best results, the data type of M should be the same as that of A. Additionally:

If sigma is None, M is symmetric positive definite
If sigma is specified, M is symmetric positive semi-definite
In buckling mode, M is symmetric indefinite.

If sigma is None, eigsh requires an operator to compute the solution of the linear equation `M * x = b`. This is done internally via a (sparse) LU decomposition for an explicit matrix M, or via an iterative solver for a general linear operator. Alternatively, the user can supply the matrix or operator Minv, which gives `x = Minv * b = M^-1 * b`.

sigma : real

Find eigenvalues near sigma using shift-invert mode. This requires an operator to compute the solution of the linear system *[A - sigma * M] x = b*, where M is the identity matrix if unspecified. This is computed internally via a (sparse) LU decomposition for explicit matrices A & M, or via an iterative solver if either A or M is a general linear operator. Alternatively, the user can supply the matrix or operator OPinv, which gives `x = OPinv * b = [A - sigma * M]^-1 * b`. Note that when sigma is specified, the keyword 'which' refers to the shifted eigenvalues `w'[i]` where:

if mode == 'normal', `w'[i] = 1 / (w[i] - sigma)`.
if mode == 'cayley', `w'[i] = (w[i] + sigma) / (w[i] - sigma)`.
if mode == 'buckling', `w'[i] = w[i] / (w[i] - sigma)`.

(see further discussion in 'mode' below)

v0 : ndarray, optional

Starting vector for iteration. Default: random

ncv : int, optional

The number of Lanczos vectors generated ncv must be greater than k and smaller than n; it is recommended that `ncv > 2*k`. Default: `min(n, max(2*k + 1, 20))`

which : str ['LM' | 'SM' | 'LA' | 'SA' | 'BE']

If A is a complex hermitian matrix, 'BE' is invalid. Which *k* eigenvectors and eigenvalues to find:

'LM' : Largest (in magnitude) eigenvalues
'SM' : Smallest (in magnitude) eigenvalues
'LA' : Largest (algebraic) eigenvalues
'SA' : Smallest (algebraic) eigenvalues
'BE' : Half (k/2) from each end of the spectrum

When k is odd, return one more (k/2+1) from the high end. When sigma != None, 'which' refers to the shifted eigenvalues `w'[i]` (see discussion in 'sigma', above). ARPACK is generally better at finding large values than small values. If small eigenvalues are desired, consider using shift-invert mode for better performance.

maxiter : int, optional

Maximum number of Arnoldi update iterations allowed Default: `n*10`

tol : float

Relative accuracy for eigenvalues (stopping criterion). The default value of 0 implies machine precision.

Minv : N x N matrix, array, sparse matrix, or LinearOperator

See notes in M, above

OPinv : N x N matrix, array, sparse matrix, or LinearOperator

See notes in sigma, above.

return_eigenvectors : bool

Return eigenvectors (True) in addition to eigenvalues

mode : string ['normal' | 'buckling' | 'cayley']

Specify strategy to use for shift-invert mode. This argument applies only for real-valued A and sigma != None. For shift-invert mode, ARPACK internally solves the eigenvalue problem `OP * x'[i] = w'[i] * B * x'[i]` and transforms the resulting Ritz vectors x'[i] and Ritz values w'[i] into the desired eigenvectors and eigenvalues of the problem `A * x[i] = w[i] * M * x[i]`. The modes are as follows:

'normal' : OP = [A - sigma * M]^-1 * M, B = M, w'[i] = 1 / (w[i] - sigma)

'buckling' : OP = [A - sigma * M]^-1 * A, B = A, w'[i] = w[i] / (w[i] - sigma)

'cayley' : OP = [A - sigma * M]^-1 * [A + sigma * M], B = M, w'[i] = (w[i] + sigma) / (w[i] - sigma)

The choice of mode will affect which eigenvalues are selected by the keyword 'which', and can also impact the stability of convergence (see [2] for a discussion)

Raises **ArpackNoConvergence**

When the requested convergence is not obtained.

The currently converged eigenvalues and eigenvectors can be found as `eigenvalues` and `eigenvectors` attributes of the exception object.

See also:

eigs eigenvalues and eigenvectors for a general (nonsymmetric) matrix A

svds singular value decomposition for a matrix A

Notes

This function is a wrapper to the ARPACK *[R19]* SSEUPD and DSEUPD functions which use the Implicitly Restarted Lanczos Method to find the eigenvalues and eigenvectors *[R20]*.

References

[R19], *[R20]*

Examples

```
>>> import scipy.sparse as sparse
>>> id = np.eye(13)
>>> vals, vecs = sparse.linalg.eigsh(id, k=6)
>>> vals
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
>>> vecs.shape
(13, 6)
```

`scipy.sparse.linalg.`**`lobpcg`**(*A, X, B=None, M=None, Y=None, tol=None, maxiter=20, largest=True, verbosityLevel=0, retLambdaHistory=False, retResidualNormsHistory=False*)

Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

LOBPCG is a preconditioned eigensolver for large symmetric positive definite (SPD) generalized eigenproblems.

Parameters **A** : {sparse matrix, dense matrix, LinearOperator}

The symmetric linear operator of the problem, usually a sparse matrix. Often called the "stiffness matrix".

X : array_like

Initial approximation to the k eigenvectors. If A has shape=(n,n) then X should have shape shape=(n,k).

B : {dense matrix, sparse matrix, LinearOperator}, optional

the right hand side operator in a generalized eigenproblem. by default, B = Identity often called the "mass matrix"

M : {dense matrix, sparse matrix, LinearOperator}, optional

preconditioner to A; by default M = Identity M should approximate the inverse of A

Y : array_like, optional

n-by-sizeY matrix of constraints, sizeY < n The iterations will be performed in the B-orthogonal complement of the column-space of Y. Y must be full rank.

Returns **w** : array

Array of k eigenvalues

v : array

An array of k eigenvectors. V has the same shape as X.

Other Parameters

tol : scalar, optional

Solver tolerance (stopping criterion) by default: tol=n*sqrt(eps)

maxiter : integer, optional

maximum number of iterations by default: maxiter=min(n,20)

largest : bool, optional

when True, solve for the largest eigenvalues, otherwise the smallest

verbosityLevel : integer, optional

controls solver output. default: verbosityLevel = 0.

retLambdaHistory : boolean, optional

whether to return eigenvalue history

retResidualNormsHistory : boolean, optional

whether to return history of residual norms

Notes

If both retLambdaHistory and retResidualNormsHistory are True, the return tuple has the following format (lambda, V, lambda history, residual norms history).

In the following `n` denotes the matrix size and `m` the number of required eigenvalues (smallest or largest).

The LOBPCG code internally solves eigenproblems of the size 3‘‘m‘‘ on every iteration by calling the "standard" dense eigensolver, so if `m` is not small enough compared to `n`, it does not make sense to call the LOBPCG code, but rather one should use the "standard" eigensolver, e.g. numpy or scipy function in this case. If one calls the LOBPCG algorithm for 5‘‘m‘‘>‘‘n‘‘, it will most likely break internally, so the code tries to call the standard function instead.

It is not that n should be large for the LOBPCG to work, but rather the ratio `n/m` should be large. It you call the LOBPCG code with `` m``=1 and ``n``=10, it should work, though ``n `` is small. The method is intended for extremely large `n/m`, see e.g., reference [28] in http://arxiv.org/abs/0705.2626

The convergence speed depends basically on two factors:

1. How well relatively separated the seeking eigenvalues are from the rest of the eigenvalues. One can try to vary `m` to make this better.
2. How well conditioned the problem is. This can be changed by using proper preconditioning. For example, a rod vibration test problem (under tests directory) is ill-conditioned for large `n`, so convergence will be slow, unless efficient preconditioning is used. For this specific problem, a good simple preconditioner function would be a linear solve for A, which is easy to code since A is tridiagonal.

Acknowledgements

lobpcg.py code was written by Robert Cimrman. Many thanks belong to Andrew Knyazev, the author of the algorithm, for lots of advice and support.

References

[R26], *[R27]*, *[R28]*

Examples

Solve A x = lambda B x with constraints and preconditioning.

```
>>> from scipy.sparse import spdiags, issparse
>>> from scipy.sparse.linalg import lobpcg, LinearOperator
>>> n = 100
>>> vals = [np.arange(n, dtype=np.float64) + 1]
>>> A = spdiags(vals, 0, n, n)
>>> A.toarray()
array([[   1.,    0.,    0., ...,    0.,    0.,    0.],
       [   0.,    2.,    0., ...,    0.,    0.,    0.],
       [   0.,    0.,    3., ...,    0.,    0.,    0.],
       ...,
       [   0.,    0.,    0., ...,   98.,    0.,    0.],
       [   0.,    0.,    0., ...,    0.,   99.,    0.],
       [   0.,    0.,    0., ...,    0.,    0.,  100.]])
```

Constraints.

```
>>> Y = np.eye(n, 3)
```

Initial guess for eigenvectors, should have linearly independent columns. Column dimension = number of requested eigenvalues.

```
>>> X = np.random.rand(n, 3)
```

Preconditioner – inverse of A (as an abstract linear operator).

```
>>> invA = spdiags([1./vals[0]], 0, n, n)
>>> def precond( x ):
...     return invA  * x
>>> M = LinearOperator(matvec=precond, shape=(n, n), dtype=float)
```

Here, `invA` could of course have been used directly as a preconditioner. Let us then solve the problem:

```
>>> eigs, vecs = lobpcg(A, X, Y=Y, M=M, tol=1e-4, maxiter=40, largest=False)
>>> eigs
array([ 4.,  5.,  6.])
```

Note that the vectors passed in Y are the eigenvectors of the 3 smallest eigenvalues. The results returned are orthogonal to those.

Singular values problems:

svds(A[, k, ncv, tol, which, v0, maxiter, ...])	Compute the largest k singular values/vectors for a sparse matrix.

`scipy.sparse.linalg.`**`svds`**(*A*, *k=6*, *ncv=None*, *tol=0*, *which='LM'*, *v0=None*, *maxiter=None*, *return_singular_vectors=True*)

Compute the largest k singular values/vectors for a sparse matrix.

Parameters **A** : {sparse matrix, LinearOperator}

Array to compute the SVD on, of shape (M, N)

k : int, optional

Number of singular values and vectors to compute. Must be 1 <= k < min(A.shape).

ncv : int, optional

The number of Lanczos vectors generated ncv must be greater than k+1 and smaller than n; it is recommended that ncv > 2*k Default: `min(n, max(2*k + 1, 20))`

tol : float, optional
Tolerance for singular values. Zero (default) means machine precision.

which : str, ['LM' | 'SM'], optional
Which *k* singular values to find:
- 'LM' : largest singular values
- 'SM' : smallest singular values

New in version 0.12.0.

v0 : ndarray, optional
Starting vector for iteration, of length min(A.shape). Should be an (approximate) left singular vector if N > M and a right singular vector otherwise. Default: random
New in version 0.12.0.

maxiter : int, optional
Maximum number of iterations.
New in version 0.12.0.

return_singular_vectors : bool or str, optional
- True: return singular vectors (True) in addition to singular values.

New in version 0.12.0.
- "u": only return the u matrix, without computing vh (if N > M).
- "vh": only return the vh matrix, without computing u (if N <= M).

New in version 0.16.0.

Returns **u** : ndarray, shape=(M, k)
Unitary matrix having left singular vectors as columns. If *return_singular_vectors* is "vh", this variable is not computed, and None is returned instead.

s : ndarray, shape=(k,)
The singular values.

vt : ndarray, shape=(k, N)
Unitary matrix having right singular vectors as rows. If *return_singular_vectors* is "u", this variable is not computed, and None is returned instead.

Notes

This is a naive implementation using ARPACK as an eigensolver on A.H * A or A * A.H, depending on which one is more efficient.

Complete or incomplete LU factorizations

`splu`(A[, permc_spec, diag_pivot_thresh, ...])	Compute the LU decomposition of a sparse, square matrix.
`spilu`(A[, drop_tol, fill_factor, drop_rule, ...])	Compute an incomplete LU decomposition for a sparse, square matrix.
`SuperLU`	LU factorization of a sparse matrix.

`scipy.sparse.linalg.`**`splu`**(*A*, *permc_spec=None*, *diag_pivot_thresh=None*, *drop_tol=None*, *relax=None*, *panel_size=None*, *options={}*)
Compute the LU decomposition of a sparse, square matrix.

Parameters **A** : sparse matrix
Sparse matrix to factorize. Should be in CSR or CSC format.

permc_spec : str, optional
How to permute the columns of the matrix for sparsity preservation. (default: 'COLAMD')

- `NATURAL`: natural ordering.
- `MMD_ATA`: minimum degree ordering on the structure of A^T A.
- `MMD_AT_PLUS_A`: minimum degree ordering on the structure of A^T+A.
- `COLAMD`: approximate minimum degree column ordering

diag_pivot_thresh : float, optional
Threshold used for a diagonal entry to be an acceptable pivot. See SuperLU user's guide for details *[R39]*

drop_tol : float, optional
(deprecated) No effect.

relax : int, optional
Expert option for customizing the degree of relaxing supernodes. See SuperLU user's guide for details *[R39]*

panel_size : int, optional
Expert option for customizing the panel size. See SuperLU user's guide for details *[R39]*

options : dict, optional
Dictionary containing additional expert options to SuperLU. See SuperLU user guide *[R39]* (section 2.4 on the 'Options' argument) for more details. For example, you can specify `options=dict(Equil=False, IterRefine='SINGLE'))` to turn equilibration off and perform a single iterative refinement.

Returns **invA** : scipy.sparse.linalg.SuperLU
Object, which has a `solve` method.

See also:

`spilu` incomplete LU decomposition

Notes

This function uses the SuperLU library.

References

[R39]

`scipy.sparse.linalg.`**`spilu`**(*A*, *drop_tol=None*, *fill_factor=None*, *drop_rule=None*, *permc_spec=None*, *diag_pivot_thresh=None*, *relax=None*, *panel_size=None*, *options=None*)

Compute an incomplete LU decomposition for a sparse, square matrix.

The resulting object is an approximation to the inverse of *A*.

Parameters **A** : (N, N) array_like
Sparse matrix to factorize

drop_tol : float, optional
Drop tolerance (0 <= tol <= 1) for an incomplete LU decomposition. (default: 1e-4)

fill_factor : float, optional
Specifies the fill ratio upper bound (>= 1.0) for ILU. (default: 10)

drop_rule : str, optional
Comma-separated string of drop rules to use. Available rules: `basic`, `prows`, `column`, `area`, `secondary`, `dynamic`, `interp`. (Default: `basic,area`)
See SuperLU documentation for details.

Remaining other options
Same as for `splu`

Returns **invA_approx** : scipy.sparse.linalg.SuperLU
Object, which has a `solve` method.

See also:

splu complete LU decomposition

Notes

To improve the better approximation to the inverse, you may need to increase *fill_factor* AND decrease *drop_tol*.

This function uses the SuperLU library.

class `scipy.sparse.linalg.`**`SuperLU`**
LU factorization of a sparse matrix.

Factorization is represented as:

```
Pr * A * Pc = L * U
```

To construct these *SuperLU* objects, call the *splu* and *spilu* functions.

Notes

New in version 0.14.0.

Examples

The LU decomposition can be used to solve matrix equations. Consider:

```
>>> import numpy as np
>>> from scipy.sparse import csc_matrix, linalg as sla
>>> A = csc_matrix([[1,2,0,4],[1,0,0,1],[1,0,2,1],[2,2,1,0.]])
```

This can be solved for a given right-hand side:

```
>>> lu = sla.splu(A)
>>> b = np.array([1, 2, 3, 4])
>>> x = lu.solve(b)
>>> A.dot(x)
array([ 1.,  2.,  3.,  4.])
```

The `lu` object also contains an explicit representation of the decomposition. The permutations are represented as mappings of indices:

```
>>> lu.perm_r
array([0, 2, 1, 3], dtype=int32)
>>> lu.perm_c
array([2, 0, 1, 3], dtype=int32)
```

The L and U factors are sparse matrices in CSC format:

```
>>> lu.L.A
array([[ 1. ,  0. ,  0. ,  0. ],
       [ 0. ,  1. ,  0. ,  0. ],
       [ 0. ,  0. ,  1. ,  0. ],
       [ 1. ,  0.5,  0.5,  1. ]])
>>> lu.U.A
array([[ 2.,  0.,  1.,  4.],
```

```
        [ 0.,  2.,  1.,  1.],
        [ 0.,  0.,  1.,  1.],
        [ 0.,  0.,  0., -5.]])
```

The permutation matrices can be constructed:

```
>>> Pr = csc_matrix((4, 4))
>>> Pr[lu.perm_r, np.arange(4)] = 1
>>> Pc = csc_matrix((4, 4))
>>> Pc[np.arange(4), lu.perm_c] = 1
```

We can reassemble the original matrix:

```
>>> (Pr.T * (lu.L * lu.U) * Pc.T).A
array([[ 1.,  2.,  0.,  4.],
       [ 1.,  0.,  0.,  1.],
       [ 1.,  0.,  2.,  1.],
       [ 2.,  2.,  1.,  0.]])
```

Attributes

shape	Shape of the original matrix as a tuple of ints.
nnz	Number of nonzero elements in the matrix.
perm_c	Permutation Pc represented as an array of indices.
perm_r	Permutation Pr represented as an array of indices.
L	Lower triangular factor with unit diagonal as a *scipy.sparse.csc_matrix*.
U	Upper triangular factor as a *scipy.sparse.csc_matrix*.

SuperLU.**shape**

Shape of the original matrix as a tuple of ints.

SuperLU.**nnz**

Number of nonzero elements in the matrix.

SuperLU.**perm_c**

Permutation Pc represented as an array of indices.

The column permutation matrix can be reconstructed via:

```
>>> Pc = np.zeros((n, n))
>>> Pc[np.arange(n), perm_c] = 1
```

SuperLU.**perm_r**

Permutation Pr represented as an array of indices.

The row permutation matrix can be reconstructed via:

```
>>> Pr = np.zeros((n, n))
>>> Pr[perm_r, np.arange(n)] = 1
```

SuperLU.**L**

Lower triangular factor with unit diagonal as a *scipy.sparse.csc_matrix*.

New in version 0.14.0.

`SuperLU.`**U**

Upper triangular factor as a *scipy.sparse.csc_matrix*.

New in version 0.14.0.

Methods

solve(rhs[, trans])	Solves linear system of equations with one or several right-hand sides.

`SuperLU.`**solve**(*rhs*[, *trans*])

Solves linear system of equations with one or several right-hand sides.

Parameters **rhs** : ndarray, shape (n,) or (n, k)

Right hand side(s) of equation

trans : {'N', 'T', 'H'}, optional

Type of system to solve:

```
'N':    A   * x == rhs  (default)
'T':    A^T * x == rhs
'H':    A^H * x == rhs
```

i.e., normal, transposed, and hermitian conjugate.

Returns **x** : ndarray, shape `rhs.shape`

Solution vector(s)

Exceptions

ArpackNoConvergence(msg, eigenvalues, ...)	ARPACK iteration did not converge
ArpackError(info[, infodict])	ARPACK error

exception `scipy.sparse.linalg.`**ArpackNoConvergence**(*msg*, *eigenvalues*, *eigenvectors*)

ARPACK iteration did not converge

Attributes

eigenvalues	(ndarray) Partial result. Converged eigenvalues.
eigenvectors	(ndarray) Partial result. Converged eigenvectors.

exception `scipy.sparse.linalg.ArpackError`(*info, infodict={'c': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size of the current Arnoldi factorization. The user is advised to check that enough workspace and array storage has been allocated.', -13: "NEV and WHICH = 'BE' are incompatible.", -12: 'IPARAM(1) must be equal to 0 or 1.', -1: 'N must be positive.', -10: 'IPARAM(7) must be 1, 2, 3.', -9: 'Starting vector is zero.', -8: 'Error return from LAPACK eigenvalue calculation;', -7: 'Length of private work array WORKL is not sufficient.', -6: "BMAT must be one of 'I' or 'G'.", -5: " WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI"', -4: 'The maximum number of Arnoldi update iterations allowed must be greater than zero.', -3: 'NCV-NEV >= 2 and less than or equal to N.', -2: 'NEV must be positive.', -11: "IPARAM(7) = 1 and BMAT = 'G' are incompatible."}, 's': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size of the current Arnoldi factorization. The user is advised to check that enough workspace and array storage has been allocated.', -13: "NEV and WHICH = 'BE' are incompatible.", -12: 'IPARAM(1) must be equal to 0 or 1.', -2: 'NEV must be positive.', -10: 'IPARAM(7) must be 1, 2, 3, 4.', -9: 'Starting vector is zero.', -8: 'Error return from LAPACK eigenvalue calculation;', -7: 'Length of private work array WORKL is not sufficient.', -6: "BMAT must be one of 'I' or 'G'.", -5: " WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI"', -4: 'The maximum number of Arnoldi update iterations allowed must be greater than zero.', -3: 'NCV-NEV >= 2 and less than or equal to N.', -1: 'N must be positive.', -11: "IPARAM(7) = 1 and BMAT = 'G' are incompatible."}, 'z': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size*

ARPACK error

Functions

aslinearoperator(A)	Return A as a LinearOperator.
bicg(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient iteration to solve `Ax = b`.
bicgstab(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient STABilized iteration to solve `Ax = b`.
cg(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient iteration to solve `Ax = b`.
cgs(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient Squared iteration to solve `Ax = b`.
eigs(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the square matrix A.
eigsh(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex hermitian matrix A.
expm(A)	Compute the matrix exponential using Pade approximation.
expm_multiply(A, B[, start, stop, num, endpoint])	Compute the action of the matrix exponential of A on B.
factorized(A)	Return a function for solving a sparse linear system, with A pre-factorized.
gmres(A, b[, x0, tol, restart, maxiter, ...])	Use Generalized Minimal RESidual iteration to solve `Ax = b`.
inv(A)	Compute the inverse of a sparse matrix
lgmres(A, b[, x0, tol, maxiter, M, ...])	Solve a matrix equation using the LGMRES algorithm.
lobpcg(A, X[, B, M, Y, tol, maxiter, ...])	Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)
lsmr(A, b[, damp, atol, btol, conlim, ...])	Iterative solver for least-squares problems.
lsqr(A, b[, damp, atol, btol, conlim, ...])	Find the least-squares solution to a large, sparse, linear system of equations.
minres(A, b[, x0, shift, tol, maxiter, ...])	Use MINimum RESidual iteration to solve Ax=b
norm(x[, ord, axis])	Norm of a sparse matrix
onenormest(A[, t, itmax, compute_v, compute_w])	Compute a lower bound of the 1-norm of a sparse matrix.
qmr(A, b[, x0, tol, maxiter, xtype, M1, M2, ...])	Use Quasi-Minimal Residual iteration to solve `Ax = b`.
spilu(A[, drop_tol, fill_factor, drop_rule, ...])	Compute an incomplete LU decomposition for a sparse, square matrix.
splu(A[, permc_spec, diag_pivot_thresh, ...])	Compute the LU decomposition of a sparse, square matrix.
spsolve(A, b[, permc_spec, use_umfpack])	Solve the sparse linear system Ax=b, where b may be a vector or a matrix.
spsolve_triangular(A, b[, lower, ...])	Solve the equation $A x = b$ for x, assuming A is a triangular matrix.
svds(A[, k, ncv, tol, which, v0, maxiter, ...])	Compute the largest k singular values/vectors for a sparse matrix.
use_solver(**kwargs)	Select default sparse direct solver to be used.

Classes

LinearOperator(dtype, shape)	Common interface for performing matrix vector products
SuperLU	LU factorization of a sparse matrix.
`Tester`	alias of `NoseTester`

Exceptions

ArpackError(info[, infodict])	ARPACK error
ArpackNoConvergence(msg, eigenvalues, ...)	ARPACK iteration did not converge
MatrixRankWarning	

Exceptions

SparseEfficiencyWarning
SparseWarning

exception scipy.sparse.**SparseEfficiencyWarning**

exception scipy.sparse.**SparseWarning**

5.21.2 Usage information

There are seven available sparse matrix types:

1. csc_matrix: Compressed Sparse Column format
2. csr_matrix: Compressed Sparse Row format
3. bsr_matrix: Block Sparse Row format
4. lil_matrix: List of Lists format
5. dok_matrix: Dictionary of Keys format
6. coo_matrix: COOrdinate format (aka IJV, triplet format)
7. dia_matrix: DIAgonal format

To construct a matrix efficiently, use either dok_matrix or lil_matrix. The lil_matrix class supports basic slicing and fancy indexing with a similar syntax to NumPy arrays. As illustrated below, the COO format may also be used to efficiently construct matrices. Despite their similarity to NumPy arrays, it is **strongly discouraged** to use NumPy functions directly on these matrices because NumPy may not properly convert them for computations, leading to unexpected (and incorrect) results. If you do want to apply a NumPy function to these matrices, first check if SciPy has its own implementation for the given sparse matrix class, or **convert the sparse matrix to a NumPy array** (e.g. using the *toarray()* method of the class) first before applying the method.

To perform manipulations such as multiplication or inversion, first convert the matrix to either CSC or CSR format. The lil_matrix format is row-based, so conversion to CSR is efficient, whereas conversion to CSC is less so.

All conversions among the CSR, CSC, and COO formats are efficient, linear-time operations.

Matrix vector product

To do a vector product between a sparse matrix and a vector simply use the matrix *dot* method, as described in its docstring:

```
>>> import numpy as np
>>> from scipy.sparse import csr_matrix
>>> A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
>>> v = np.array([1, 0, -1])
```

```
>>> A.dot(v)
array([ 1, -3, -1], dtype=int64)
```

Warning: As of NumPy 1.7, *np.dot* is not aware of sparse matrices, therefore using it will result on unexpected results or errors. The corresponding dense array should be obtained first instead:

```
>>> np.dot(A.toarray(), v)
array([ 1, -3, -1], dtype=int64)
```

but then all the performance advantages would be lost.

The CSR format is specially suitable for fast matrix vector products.

Example 1

Construct a 1000x1000 lil_matrix and add some values to it:

```
>>> from scipy.sparse import lil_matrix
>>> from scipy.sparse.linalg import spsolve
>>> from numpy.linalg import solve, norm
>>> from numpy.random import rand
```

```
>>> A = lil_matrix((1000, 1000))
>>> A[0, :100] = rand(100)
>>> A[1, 100:200] = A[0, :100]
>>> A.setdiag(rand(1000))
```

Now convert it to CSR format and solve A x = b for x:

```
>>> A = A.tocsr()
>>> b = rand(1000)
>>> x = spsolve(A, b)
```

Convert it to a dense matrix and solve, and check that the result is the same:

```
>>> x_ = solve(A.toarray(), b)
```

Now we can compute norm of the error with:

```
>>> err = norm(x-x_)
>>> err < 1e-10
True
```

It should be small :)

Example 2

Construct a matrix in COO format:

```
>>> from scipy import sparse
>>> from numpy import array
>>> I = array([0,3,1,0])
>>> J = array([0,3,1,2])
```

```
>>> V = array([4,5,7,9])
>>> A = sparse.coo_matrix((V,(I,J)),shape=(4,4))
```

Notice that the indices do not need to be sorted.

Duplicate (i,j) entries are summed when converting to CSR or CSC.

```
>>> I = array([0,0,1,3,1,0,0])
>>> J = array([0,2,1,3,1,0,0])
>>> V = array([1,1,1,1,1,1,1])
>>> B = sparse.coo_matrix((V,(I,J)),shape=(4,4)).tocsr()
```

This is useful for constructing finite-element stiffness and mass matrices.

Further Details

CSR column indices are not necessarily sorted. Likewise for CSC row indices. Use the .sorted_indices() and .sort_indices() methods when sorted indices are required (e.g. when passing data to other libraries).

5.22 Sparse linear algebra (`scipy.sparse.linalg`)

5.22.1 Abstract linear operators

LinearOperator(dtype, shape)	Common interface for performing matrix vector products
aslinearoperator(A)	Return A as a LinearOperator.

class `scipy.sparse.linalg.`**`LinearOperator`**(*dtype*, *shape*)

Common interface for performing matrix vector products

Many iterative methods (e.g. cg, gmres) do not need to know the individual entries of a matrix to solve a linear system A*x=b. Such solvers only require the computation of matrix vector products, A*v where v is a dense vector. This class serves as an abstract interface between iterative solvers and matrix-like objects.

To construct a concrete LinearOperator, either pass appropriate callables to the constructor of this class, or subclass it.

A subclass must implement either one of the methods `_matvec` and `_matmat`, and the attributes/properties `shape` (pair of integers) and `dtype` (may be None). It may call the `__init__` on this class to have these attributes validated. Implementing `_matvec` automatically implements `_matmat` (using a naive algorithm) and vice-versa.

Optionally, a subclass may implement `_rmatvec` or `_adjoint` to implement the Hermitian adjoint (conjugate transpose). As with `_matvec` and `_matmat`, implementing either `_rmatvec` or `_adjoint` implements the other automatically. Implementing `_adjoint` is preferable; `_rmatvec` is mostly there for backwards compatibility.

Parameters

shape : tuple
    Matrix dimensions (M,N).

matvec : callable f(v)
    Returns returns A * v.

rmatvec : callable f(v)
    Returns A^H * v, where A^H is the conjugate transpose of A.

matmat : callable f(V)

Returns A * V, where V is a dense matrix with dimensions (N,K).

dtype : dtype

Data type of the matrix.

See also:

aslinearoperator
Construct LinearOperators

Notes

The user-defined matvec() function must properly handle the case where v has shape (N,) as well as the (N,1) case. The shape of the return type is handled internally by LinearOperator.

LinearOperator instances can also be multiplied, added with each other and exponentiated, all lazily: the result of these operations is always a new, composite LinearOperator, that defers linear operations to the original operators and combines the results.

Examples

```
>>> import numpy as np
>>> from scipy.sparse.linalg import LinearOperator
>>> def mv(v):
...     return np.array([2*v[0], 3*v[1]])
...
>>> A = LinearOperator((2,2), matvec=mv)
>>> A
<2x2 _CustomLinearOperator with dtype=float64>
>>> A.matvec(np.ones(2))
array([ 2.,  3.])
>>> A * np.ones(2)
array([ 2.,  3.])
```

Attributes

args	(tuple) For linear operators describing products etc. of other linear operators, the operands of the binary operation.

Methods

`__call__`(x)	
`adjoint`()	Hermitian adjoint.
`dot`(x)	Matrix-matrix or matrix-vector multiplication.
`matmat`(X)	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`transpose`()	Transpose this linear operator.

LinearOperator.**__call__**(*x*)

LinearOperator.**adjoint**()
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns **A_H** : LinearOperator
Hermitian adjoint of self.

LinearOperator.**dot**(*x*)
Matrix-matrix or matrix-vector multiplication.

Parameters **x** : array_like
1-d or 2-d array, representing a vector or matrix.

Returns **Ax** : array
1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

LinearOperator.**matmat**(*X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters **X** : {matrix, ndarray}
An array with shape (N,K).

Returns **Y** : {matrix, ndarray}
A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

LinearOperator.**matvec**(*x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}
An array with shape (N,) or (N,1).

Returns **y** : {matrix, ndarray}
A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

LinearOperator.**rmatvec**(*x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters **x** : {matrix, ndarray}
An array with shape (M,) or (M,1).

Returns **y** : {matrix, ndarray}
A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

LinearOperator.**transpose**()
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

`scipy.sparse.linalg.`**`aslinearoperator`**`(A)`

Return A as a LinearOperator.

'A' may be any of the following types:

- ndarray
- matrix
- sparse matrix (e.g. csr_matrix, lil_matrix, etc.)
- LinearOperator
- An object with .shape and .matvec attributes

See the LinearOperator documentation for additional information.

Examples

```
>>> from scipy.sparse.linalg import aslinearoperator
>>> M = np.array([[1,2,3],[4,5,6]], dtype=np.int32)
>>> aslinearoperator(M)
<2x3 MatrixLinearOperator with dtype=int32>
```

5.22.2 Matrix Operations

inv(A)	Compute the inverse of a sparse matrix
expm(A)	Compute the matrix exponential using Pade approximation.
expm_multiply(A, B[, start, stop, num, endpoint])	Compute the action of the matrix exponential of A on B.

`scipy.sparse.linalg.`**`inv`**`(A)`

Compute the inverse of a sparse matrix

Parameters **A** : (M,M) ndarray or sparse matrix

square matrix to be inverted

Returns **Ainv** : (M,M) ndarray or sparse matrix

inverse of *A*

Notes

This computes the sparse inverse of *A*. If the inverse of *A* is expected to be non-sparse, it will likely be faster to convert *A* to dense and use scipy.linalg.inv.

New in version 0.12.0.

`scipy.sparse.linalg.`**`expm`**`(A)`

Compute the matrix exponential using Pade approximation.

Parameters **A** : (M,M) array_like or sparse matrix

2D Array or Matrix (sparse or dense) to be exponentiated

Returns **expA** : (M,M) ndarray

Matrix exponential of *A*

Notes

This is algorithm (6.1) which is a simplification of algorithm (5.1).

New in version 0.12.0.

References

[R331]

`scipy.sparse.linalg.`**`expm_multiply`**(*A*, *B*, *start=None*, *stop=None*, *num=None*, *endpoint=None*)

Compute the action of the matrix exponential of A on B.

Parameters

A : transposable linear operator
    The operator whose exponential is of interest.

B : ndarray
    The matrix or vector to be multiplied by the matrix exponential of A.

start : scalar, optional
    The starting time point of the sequence.

stop : scalar, optional
    The end time point of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced time points, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : int, optional
    Number of time points to use.

endpoint : bool, optional
    If True, *stop* is the last time point. Otherwise, it is not included.

Returns

expm_A_B : ndarray
    The result of the action $e^{t_k A}B$.

Notes

The optional arguments defining the sequence of evenly spaced time points are compatible with the arguments of `numpy.linspace`.

The output ndarray shape is somewhat complicated so I explain it here. The ndim of the output could be either 1, 2, or 3. It would be 1 if you are computing the expm action on a single vector at a single time point. It would be 2 if you are computing the expm action on a vector at multiple time points, or if you are computing the expm action on a matrix at a single time point. It would be 3 if you want the action on a matrix with multiple columns at multiple time points. If multiple time points are requested, expm_A_B[0] will always be the action of the expm at the first time point, regardless of whether the action is on a vector or a matrix.

References

[R332], *[R333]*

5.22.3 Matrix norms

norm(x[, ord, axis])	Norm of a sparse matrix
onenormest(A[, t, itmax, compute_v, compute_w])	Compute a lower bound of the 1-norm of a sparse matrix.

`scipy.sparse.linalg.`**`norm`**(*x*, *ord=None*, *axis=None*)

Norm of a sparse matrix

This function is able to return one of seven different matrix norms, depending on the value of the `ord` parameter.

Parameters

x : a sparse matrix
    Input sparse matrix.

ord : {non-zero int, inf, -inf, ‘fro’}, optional
    Order of the norm (see table under `Notes`). inf means numpy’s *inf* object.

axis : {int, 2-tuple of ints, None}, optional

If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned.

Returns **n** : float or ndarray

Notes

Some of the ord are not implemented because some associated functions like, _multi_svd_norm, are not yet available for sparse matrix.

This docstring is modified based on numpy.linalg.norm. https://github.com/numpy/numpy/blob/master/numpy/linalg/linalg.py

The following norms can be calculated:

ord	norm for sparse matrices
None	Frobenius norm
'fro'	Frobenius norm
inf	max(sum(abs(x), axis=1))
-inf	min(sum(abs(x), axis=1))
0	abs(x).sum(axis=axis)
1	max(sum(abs(x), axis=0))
-1	min(sum(abs(x), axis=0))
2	Not implemented
-2	Not implemented
other	Not implemented

The Frobenius norm is given by *[R344]*:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

References

[R344]

Examples

```
>>> from scipy.sparse import *
>>> import numpy as np
>>> from scipy.sparse.linalg import norm
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
>>> b = a.reshape((3, 3))
>>> b
array([[-4, -3, -2],
       [-1, 0, 1],
       [ 2, 3, 4]])
```

```
>>> b = csr_matrix(b)
>>> norm(b)
7.745966692414834
>>> norm(b, 'fro')
7.745966692414834
>>> norm(b, np.inf)
9
>>> norm(b, -np.inf)
```

```
2
>>> norm(b, 1)
7
>>> norm(b, -1)
6
```

scipy.sparse.linalg.**onenormest**(*A*, *t=2*, *itmax=5*, *compute_v=False*, *compute_w=False*)

Compute a lower bound of the 1-norm of a sparse matrix.

Parameters

A : ndarray or other linear operator
: A linear operator that can be transposed and that can produce matrix products.

t : int, optional
: A positive parameter controlling the tradeoff between accuracy versus time and memory usage. Larger values take longer and use more memory but give more accurate output.

itmax : int, optional
: Use at most this many iterations.

compute_v : bool, optional
: Request a norm-maximizing linear operator input vector if True.

compute_w : bool, optional
: Request a norm-maximizing linear operator output vector if True.

Returns

est : float
: An underestimate of the 1-norm of the sparse matrix.

v : ndarray, optional
: The vector such that ||Av||_1 == est*||v||_1. It can be thought of as an input to the linear operator that gives an output with particularly large norm.

w : ndarray, optional
: The vector Av which has relatively large 1-norm. It can be thought of as an output of the linear operator that is relatively large in norm compared to the input.

Notes

This is algorithm 2.4 of [1].

In [2] it is described as follows. "This algorithm typically requires the evaluation of about 4t matrix-vector products and almost invariably produces a norm estimate (which is, in fact, a lower bound on the norm) correct to within a factor 3."

New in version 0.13.0.

References

[R345], *[R346]*

5.22.4 Solving linear problems

Direct methods for linear equation systems:

spsolve(A, b[, permc_spec, use_umfpack])	Solve the sparse linear system Ax=b, where b may be a vector or a matrix.
spsolve_triangular(A, b[, lower, ...])	Solve the equation $A x = b$ for x, assuming A is a triangular matrix.

Continued on next page

Table 5.196 – continued from previous page

factorized(A)	Return a function for solving a sparse linear system, with A pre-factorized.
MatrixRankWarning	
use_solver(**kwargs)	Select default sparse direct solver to be used.

scipy.sparse.linalg.**spsolve**(*A*, *b*, *permc_spec=None*, *use_umfpack=True*)

Solve the sparse linear system Ax=b, where b may be a vector or a matrix.

Parameters

A : ndarray or sparse matrix
: The square matrix A will be converted into CSC or CSR form

b : ndarray or sparse matrix
: The matrix or vector representing the right hand side of the equation. If a vector, b.shape must be (n,) or (n, 1).

permc_spec : str, optional
: How to permute the columns of the matrix for sparsity preservation. (default: 'COLAMD')
- NATURAL: natural ordering.
- MMD_ATA: minimum degree ordering on the structure of A^T A.
- MMD_AT_PLUS_A: minimum degree ordering on the structure of A^T+A.
- COLAMD: approximate minimum degree column ordering

use_umfpack : bool, optional
: if True (default) then use umfpack for the solution. This is only referenced if b is a vector and scikit-umfpack is installed.

Returns

x : ndarray or sparse matrix
: the solution of the sparse linear equation. If b is a vector, then x is a vector of size A.shape[1] If b is a matrix, then x is a matrix of size (A.shape[1], b.shape[1])

Notes

For solving the matrix expression AX = B, this solver assumes the resulting matrix X is sparse, as is often the case for very sparse inputs. If the resulting X is dense, the construction of this sparse result will be relatively expensive. In that case, consider converting A to a dense matrix and using scipy.linalg.solve or its variants.

scipy.sparse.linalg.**spsolve_triangular**(*A*, *b*, *lower=True*, *overwrite_A=False*, *overwrite_b=False*)

Solve the equation $A x = b$ for x, assuming A is a triangular matrix.

Parameters

A : (M, M) sparse matrix
: A sparse square triangular matrix. Should be in CSR format.

b : (M,) or (M, N) array_like
: Right-hand side matrix in $A x = b$

lower : bool, optional
: Whether *A* is a lower or upper triangular matrix. Default is lower triangular matrix.

overwrite_A : bool, optional
: Allow changing *A*. The indices of *A* are going to be sorted and zero entries are going to be removed. Enabling gives a performance gain. Default is False.

overwrite_b : bool, optional
: Allow overwriting data in *b*. Enabling gives a performance gain. Default is False.

Returns

x : (M,) or (M, N) ndarray
: Solution to the system $A x = b$. Shape of return matches shape of *b*.

Raises

LinAlgError
: If *A* is singular or not triangular.

ValueError
If shape of *A* or shape of *b* do not match the requirements.

Notes

New in version 0.19.0.

`scipy.sparse.linalg.factorized(A)`
Return a function for solving a sparse linear system, with A pre-factorized.

Parameters **A** : (N, N) array_like
Input.

Returns **solve** : callable
To solve the linear system of equations given in *A*, the *solve* callable should be passed an ndarray of shape (N,).

Examples

```
>>> from scipy.sparse.linalg import factorized
>>> A = np.array([[ 3. ,   2. , -1. ],
...               [ 2. , -2. ,   4. ],
...               [-1. ,  0.5, -1. ]])
>>> solve = factorized(A) # Makes LU decomposition.
>>> rhs1 = np.array([1, -2, 0])
>>> solve(rhs1) # Uses the LU factors.
array([ 1., -2., -2.])
```

exception `scipy.sparse.linalg.MatrixRankWarning`

`scipy.sparse.linalg.use_solver(**kwargs)`
Select default sparse direct solver to be used.

Parameters **useUmfpack** : bool, optional
Use UMFPACK over SuperLU. Has effect only if scikits.umfpack is installed. Default: True

Notes

The default sparse solver is umfpack when available (scikits.umfpack is installed). This can be changed by passing useUmfpack = False, which then causes the always present SuperLU based solver to be used.

Umfpack requires a CSR/CSC matrix to have sorted column/row indices. If sure that the matrix fulfills this, pass `assumeSortedIndices=True` to gain some speed.

Iterative methods for linear equation systems:

bicg(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient iteration to solve `Ax = b`.
bicgstab(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient STABilized iteration to solve `Ax = b`.
cg(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient iteration to solve `Ax = b`.
cgs(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient Squared iteration to solve `Ax = b`.
gmres(A, b[, x0, tol, restart, maxiter, ...])	Use Generalized Minimal RESidual iteration to solve `Ax = b`.
lgmres(A, b[, x0, tol, maxiter, M, ...])	Solve a matrix equation using the LGMRES algorithm.
minres(A, b[, x0, shift, tol, maxiter, ...])	Use MINimum RESidual iteration to solve Ax=b
qmr(A, b[, x0, tol, maxiter, xtype, M1, M2, ...])	Use Quasi-Minimal Residual iteration to solve `Ax = b`.

`scipy.sparse.linalg.`**`bicg`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use BIConjugate Gradient iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
: The real or complex N-by-N matrix of the linear system. It is required that the linear operator can produce `Ax` and `A^T x`.

b : {array, matrix}
: Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}
: The converged solution.

info : integer
: ***Provides convergence information:***
  0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}
: Starting guess for the solution.

tol : float
: Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer
: Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}
: Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function
: User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
: This parameter is deprecated – avoid using it.
  The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`bicgstab`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use BIConjugate Gradient STABilized iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
: The real or complex N-by-N matrix of the linear system.

b : {array, matrix}
: Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}
: The converged solution.

info : integer
: ***Provides convergence information:***
  0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}

This parameter is deprecated – avoid using it.

The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`cg`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use Conjugate Gradient iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real or complex N-by-N matrix of the linear system. `A` must represent a hermitian, positive definite matrix.

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
This parameter is deprecated – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`cgs`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*)

Use Conjugate Gradient Squared iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
The real-valued N-by-N matrix of the linear system.

b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}
The converged solution.

info : integer
Provides convergence information:
0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}
Starting guess for the solution.

tol : float
Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer
Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}
Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
This parameter is deprecated – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

`scipy.sparse.linalg.`**`gmres`**(*A*, *b*, *x0=None*, *tol=1e-05*, *restart=None*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*, *restrt=None*)

Use Generalized Minimal RESidual iteration to solve `Ax = b`.

Parameters **A** : {sparse matrix, dense matrix, LinearOperator}
The real or complex N-by-N matrix of the linear system.

b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

Returns **x** : {array, matrix}
The converged solution.

info : int
Provides convergence information:

- 0 : successful exit
- >0 : convergence to tolerance not achieved, number of iterations
- <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}
Starting guess for the solution (a vector of zeros by default).

tol : float
Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

restart : int, optional
Number of iterations between restarts. Larger values increase iteration cost, but may be necessary for convergence. Default is 20.

maxiter : int, optional
Maximum number of iterations (restart cycles). Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

xtype : {'f','d','F','D'}
This parameter is DEPRECATED — avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

M : {sparse matrix, dense matrix, LinearOperator}
Inverse of the preconditioner of A. M should approximate the inverse of A and be easy to solve for (see Notes). Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance. By default, no preconditioner is used.

callback : function
User-supplied function to call after each iteration. It is called as callback(rk), where rk is the current residual vector.

restrt : int, optional
DEPRECATED - use *restart* instead.

See also:

`LinearOperator`

Notes

A preconditioner, P, is chosen such that P is close to A but easy to solve for. The preconditioner parameter required by this routine is `M = P^-1`. The inverse should preferably not be calculated explicitly. Rather, use the following template to produce M:

```
# Construct a linear operator that computes P^-1 * x.
import scipy.sparse.linalg as spla
M_x = lambda x: spla.spsolve(P, x)
M = spla.LinearOperator((n, n), M_x)
```

`scipy.sparse.linalg.`**`lgmres`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=1000*, *M=None*, *callback=None*, *inner_m=30*, *outer_k=3*, *outer_v=None*, *store_outer_Av=True*)

Solve a matrix equation using the LGMRES algorithm.

The LGMRES algorithm *[R334] [R335]* is designed to avoid some problems in the convergence in restarted GMRES, and often converges in fewer iterations.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
    The real or complex N-by-N matrix of the linear system.

b : {array, matrix}
    Right hand side of the linear system. Has shape (N,) or (N,1).

x0 : {array, matrix}
    Starting guess for the solution.

tol : float, optional
    Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : int, optional
    Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}, optional
    Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function, optional
    User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

inner_m : int, optional
    Number of inner GMRES iterations per each outer iteration.

outer_k : int, optional
    Number of vectors to carry between inner GMRES iterations. According to *[R334]*, good values are in the range of 1...3. However, note that if you want to use the additional vectors to accelerate solving multiple similar problems, larger values may be beneficial.

outer_v : list of tuples, optional
    List containing tuples `(v, Av)` of vectors and corresponding matrix-vector products, used to augment the Krylov subspace, and carried between inner GMRES iterations. The element `Av` can be *None* if the matrix-vector product should be re-evaluated. This parameter is modified in-place by *`lgmres`*, and can be used to pass "guess" vectors in and out of the algorithm when solving similar problems.

store_outer_Av : bool, optional
    Whether LGMRES should store also A*v in addition to vectors *v* in the *outer_v* list. Default is True.

Returns

x : array or matrix
    The converged solution.

info : int
    Provides convergence information:

- 0 : successful exit
- >0 : convergence to tolerance not achieved, number of iterations

• <0 : illegal input or breakdown

Notes

The LGMRES algorithm *[R334] [R335]* is designed to avoid the slowing of convergence in restarted GMRES, due to alternating residual vectors. Typically, it often outperforms GMRES(m) of comparable memory requirements by some measure, or at least is not much worse.

Another advantage in this algorithm is that you can supply it with 'guess' vectors in the *outer_v* argument that augment the Krylov subspace. If the solution lies close to the span of these vectors, the algorithm converges faster. This can be useful if several very similar matrices need to be inverted one after another, such as in Newton-Krylov iteration where the Jacobian matrix often changes little in the nonlinear steps.

References

[R334], *[R335]*

scipy.sparse.linalg.**minres**(*A*, *b*, *x0=None*, *shift=0.0*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M=None*, *callback=None*, *show=False*, *check=False*)

Use MINimum RESidual iteration to solve Ax=b

MINRES minimizes norm(A*x - b) for a real symmetric matrix A. Unlike the Conjugate Gradient method, A can be indefinite or singular.

If shift != 0 then the method solves (A - shift*I)x = b

Parameters

A : {sparse matrix, dense matrix, LinearOperator}

The real symmetric N-by-N matrix of the linear system

b : {array, matrix}

Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}

The converged solution.

info : integer

Provides convergence information:

0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}

Starting guess for the solution.

tol : float

Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer

Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M : {sparse matrix, dense matrix, LinearOperator}

Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning dramatically improves the rate of convergence, which implies that fewer iterations are needed to reach a given error tolerance.

callback : function

User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}

This parameter is deprecated – avoid using it.

The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A

does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

Notes

THIS FUNCTION IS EXPERIMENTAL AND SUBJECT TO CHANGE!

References

Solution of sparse indefinite systems of linear equations,
C. C. Paige and M. A. Saunders (1975), SIAM J. Numer. Anal. 12(4), pp. 617-629. http://www.stanford.edu/group/SOL/software/minres.html

This file is a translation of the following MATLAB implementation:
http://www.stanford.edu/group/SOL/software/minres/matlab/

`scipy.sparse.linalg.`**`qmr`**(*A*, *b*, *x0=None*, *tol=1e-05*, *maxiter=None*, *xtype=None*, *M1=None*, *M2=None*, *callback=None*)

Use Quasi-Minimal Residual iteration to solve `Ax = b`.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
The real-valued N-by-N matrix of the linear system. It is required that the linear operator can produce `Ax` and `A^T x`.

b : {array, matrix}
Right hand side of the linear system. Has shape (N,) or (N,1).

Returns

x : {array, matrix}
The converged solution.

info : integer
Provides convergence information:
0 : successful exit >0 : convergence to tolerance not achieved, number of iterations <0 : illegal input or breakdown

Other Parameters

x0 : {array, matrix}
Starting guess for the solution.

tol : float
Tolerance to achieve. The algorithm terminates when either the relative or the absolute residual is below *tol*.

maxiter : integer
Maximum number of iterations. Iteration will stop after maxiter steps even if the specified tolerance has not been achieved.

M1 : {sparse matrix, dense matrix, LinearOperator}
Left preconditioner for A.

M2 : {sparse matrix, dense matrix, LinearOperator}
Right preconditioner for A. Used together with the left preconditioner M1. The matrix M1*A*M2 should have better conditioned than A alone.

callback : function
User-supplied function to call after each iteration. It is called as callback(xk), where xk is the current solution vector.

xtype : {'f','d','F','D'}
This parameter is DEPRECATED – avoid using it.
The type of the result. If None, then it will be determined from A.dtype.char and b. If A does not have a typecode method then it will compute A.matvec(x0) to get a typecode. To save the extra computation when A does not have a typecode attribute use xtype=0 for the same type as b or use xtype='f','d','F',or 'D'. This parameter has been superseded by LinearOperator.

See also:

LinearOperator

Iterative methods for least-squares problems:

lsqr(A, b[, damp, atol, btol, conlim, ...])	Find the least-squares solution to a large, sparse, linear system of equations.
lsmr(A, b[, damp, atol, btol, conlim, ...])	Iterative solver for least-squares problems.

scipy.sparse.linalg.**lsqr**(*A*, *b*, *damp=0.0*, *atol=1e-08*, *btol=1e-08*, *conlim=100000000.0*, *iter_lim=None*, *show=False*, *calc_var=False*)

Find the least-squares solution to a large, sparse, linear system of equations.

The function solves `Ax = b` or `min ||b - Ax||^2` or `min ||Ax - b||^2 + d^2 ||x||^2`.

The matrix A may be square or rectangular (over-determined or under-determined), and may have any rank.

```
1. Unsymmetric equations --    solve  A*x = b

2. Linear least squares  --    solve  A*x = b
                               in the least-squares sense

3. Damped least squares  --    solve  (   A    )*x = ( b )
                                      ( damp*I )     ( 0 )
                               in the least-squares sense
```

Parameters

A : {sparse matrix, ndarray, LinearOperator}
: Representation of an m-by-n matrix. It is required that the linear operator can produce `Ax` and `A^T x`.

b : array_like, shape (m,)
: Right-hand side vector `b`.

damp : float
: Damping coefficient.

atol, btol : float, optional
: Stopping tolerances. If both are 1.0e-9 (say), the final residual norm should be accurate to about 9 digits. (The final x will usually have fewer correct digits, depending on cond(A) and the size of damp.)

conlim : float, optional
: Another stopping tolerance. lsqr terminates if an estimate of `cond(A)` exceeds *conlim*. For compatible systems `Ax = b`, *conlim* could be as large as 1.0e+12 (say). For least-squares problems, conlim should be less than 1.0e+8. Maximum precision can be obtained by setting `atol = btol = conlim = zero`, but the number of iterations may then be excessive.

iter_lim : int, optional
: Explicit limitation on number of iterations (for safety).

show : bool, optional
: Display an iteration log.

calc_var : bool, optional
: Whether to estimate diagonals of `(A'A + damp^2*I)^{-1}`.

Returns

x : ndarray of float
: The final solution.

istop : int
: Gives the reason for termination. 1 means x is an approximate solution to Ax = b. 2 means x approximately solves the least-squares problem.

itn : int

Iteration number upon termination.

r1norm : float

norm(r), where r = b - Ax.

r2norm : float

sqrt(norm(r)^2 + damp^2 * norm(x)^2). Equal to *r1norm* if damp == 0.

anorm : float

Estimate of Frobenius norm of Abar = [[A]; [damp*I]].

acond : float

Estimate of cond(Abar).

arnorm : float

Estimate of norm(A'*r - damp^2*x).

xnorm : float

norm(x)

var : ndarray of float

If calc_var is True, estimates all diagonals of (A'A)^{-1} (if damp == 0) or more generally (A'A + damp^2*I)^{-1}. This is well defined if A has full column rank or damp > 0. (Not sure what var means if rank(A) < n and damp = 0.)

Notes

LSQR uses an iterative method to approximate the solution. The number of iterations required to reach a certain accuracy depends strongly on the scaling of the problem. Poor scaling of the rows or columns of A should therefore be avoided where possible.

For example, in problem 1 the solution is unaltered by row-scaling. If a row of A is very small or large compared to the other rows of A, the corresponding row of (A b) should be scaled up or down.

In problems 1 and 2, the solution x is easily recovered following column-scaling. Unless better information is known, the nonzero columns of A should be scaled so that they all have the same Euclidean norm (e.g., 1.0).

In problem 3, there is no freedom to re-scale if damp is nonzero. However, the value of damp should be assigned only after attention has been paid to the scaling of A.

The parameter damp is intended to help regularize ill-conditioned systems, by preventing the true solution from being very large. Another aid to regularization is provided by the parameter acond, which may be used to terminate iterations before the computed solution becomes very large.

If some initial estimate x0 is known and if damp == 0, one could proceed as follows:

1. Compute a residual vector r0 = b - A*x0.
2. Use LSQR to solve the system A*dx = r0.
3. Add the correction dx to obtain a final solution x = x0 + dx.

This requires that x0 be available before and after the call to LSQR. To judge the benefits, suppose LSQR takes k1 iterations to solve A*x = b and k2 iterations to solve A*dx = r0. If x0 is "good", norm(r0) will be smaller than norm(b). If the same stopping tolerances atol and btol are used for each system, k1 and k2 will be similar, but the final solution x0 + dx should be more accurate. The only way to reduce the total work is to use a larger stopping tolerance for the second system. If some value btol is suitable for A*x = b, the larger value btol*norm(b)/norm(r0) should be suitable for A*dx = r0.

Preconditioning is another way to reduce the number of iterations. If it is possible to solve a related system M*x = b efficiently, where M approximates A in some helpful way (e.g. M - A has low rank or its elements are small relative to those of A), LSQR may converge more rapidly on the system A*M(inverse)*z = b, after which x can be recovered by solving M*x = z.

If A is symmetric, LSQR should not be used!

Alternatives are the symmetric conjugate-gradient method (cg) and/or SYMMLQ. SYMMLQ is an implementation of symmetric cg that applies to any symmetric A and will converge more rapidly than LSQR. If A is positive definite, there are other implementations of symmetric cg that require slightly less work per iteration than SYMMLQ (but will take the same number of iterations).

References

[R341], *[R342]*, *[R343]*

scipy.sparse.linalg.**lsmr**(*A*, *b*, *damp=0.0*, *atol=1e-06*, *btol=1e-06*, *conlim=100000000.0*, *maxiter=None*, *show=False*)

Iterative solver for least-squares problems.

lsmr solves the system of linear equations `Ax = b`. If the system is inconsistent, it solves the least-squares problem `min ||b - Ax||_2`. A is a rectangular matrix of dimension m-by-n, where all cases are allowed: m = n, m > n, or m < n. B is a vector of length m. The matrix A may be dense or sparse (usually sparse).

Parameters

A : {matrix, sparse matrix, ndarray, LinearOperator}
Matrix A in the linear system.

b : array_like, shape (m,)
Vector b in the linear system.

damp : float
Damping factor for regularized least-squares. *lsmr* solves the regularized least-squares problem:

```
min ||(b) - (   A    )x||
    ||(0)     (damp*I) ||_2
```

where damp is a scalar. If damp is None or 0, the system is solved without regularization.

atol, btol : float, optional
Stopping tolerances. *lsmr* continues iterations until a certain backward error estimate is smaller than some quantity depending on atol and btol. Let `r = b - Ax` be the residual vector for the current approximate solution `x`. If `Ax = b` seems to be consistent, `lsmr` terminates when `norm(r) <= atol * norm(A) * norm(x) + btol * norm(b)`. Otherwise, lsmr terminates when `norm(A^{T} r) <= atol * norm(A) * norm(r)`. If both tolerances are 1.0e-6 (say), the final `norm(r)` should be accurate to about 6 digits. (The final x will usually have fewer correct digits, depending on `cond(A)` and the size of LAMBDA.) If *atol* or *btol* is None, a default value of 1.0e-6 will be used. Ideally, they should be estimates of the relative error in the entries of A and B respectively. For example, if the entries of *A* have 7 correct digits, set atol = 1e-7. This prevents the algorithm from doing unnecessary work beyond the uncertainty of the input data.

conlim : float, optional
lsmr terminates if an estimate of `cond(A)` exceeds *conlim*. For compatible systems `Ax = b`, conlim could be as large as 1.0e+12 (say). For least-squares problems, *conlim* should be less than 1.0e+8. If *conlim* is None, the default value is 1e+8. Maximum precision can be obtained by setting `atol = btol = conlim = 0`, but the number of iterations may then be excessive.

maxiter : int, optional
lsmr terminates if the number of iterations reaches *maxiter*. The default is `maxiter = min(m, n)`. For ill-conditioned systems, a larger value of *maxiter* may be needed.

show : bool, optional

Print iterations logs if `show=True`.

Returns **x** : ndarray of float

Least-square solution returned.

istop : int

istop gives the reason for stopping:

```
istop   = 0 means x=0 is a solution.
        = 1 means x is an approximate solution to A*x␣
↪= B,
          according to atol and btol.
        = 2 means x approximately solves the least-
↪squares problem
          according to atol.
        = 3 means COND(A) seems to be greater than␣
↪CONLIM.
        = 4 is the same as 1 with atol = btol = eps␣
↪(machine
          precision)
        = 5 is the same as 2 with atol = eps.
        = 6 is the same as 3 with CONLIM = 1/eps.
        = 7 means ITN reached maxiter before the other␣
↪stopping
          conditions were satisfied.
```

itn : int

Number of iterations used.

normr : float

`norm(b-Ax)`

normar : float

`norm(A^T (b - Ax))`

norma : float

`norm(A)`

conda : float

Condition number of A.

normx : float

`norm(x)`

Notes

New in version 0.11.0.

References

[R339], [R340]

5.22.5 Matrix factorizations

Eigenvalue problems:

eigs(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the square matrix A.
eigsh(A[, k, M, sigma, which, v0, ncv, ...])	Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex hermitian matrix A.
lobpcg(A, X[, B, M, Y, tol, maxiter, ...])	Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

`scipy.sparse.linalg.eigs`(*A*, *k=6*, *M=None*, *sigma=None*, *which='LM'*, *v0=None*, *ncv=None*, *maxiter=None*, *tol=0*, *return_eigenvectors=True*, *Minv=None*, *OPinv=None*, *OPpart=None*)

Find k eigenvalues and eigenvectors of the square matrix A.

Solves `A * x[i] = w[i] * x[i]`, the standard eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i].

If M is specified, solves `A * x[i] = w[i] * M * x[i]`, the generalized eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i]

Parameters

A : ndarray, sparse matrix or LinearOperator

An array, sparse matrix, or LinearOperator representing the operation `A * x`, where A is a real or complex square matrix.

k : int, optional

The number of eigenvalues and eigenvectors desired. *k* must be smaller than N-1. It is not possible to compute all eigenvectors of a matrix.

M : ndarray, sparse matrix or LinearOperator, optional

An array, sparse matrix, or LinearOperator representing the operation M*x for the generalized eigenvalue problem

A * x = w * M * x.

M must represent a real, symmetric matrix if A is real, and must represent a complex, hermitian matrix if A is complex. For best results, the data type of M should be the same as that of A. Additionally:

If *sigma* is None, M is positive definite

If sigma is specified, M is positive semi-definite

If sigma is None, eigs requires an operator to compute the solution of the linear equation `M * x = b`. This is done internally via a (sparse) LU decomposition for an explicit matrix M, or via an iterative solver for a general linear operator. Alternatively, the user can supply the matrix or operator Minv, which gives `x = Minv * b = M^-1 * b`.

sigma : real or complex, optional

Find eigenvalues near sigma using shift-invert mode. This requires an operator to compute the solution of the linear system `[A - sigma * M] * x = b`, where M is the identity matrix if unspecified. This is computed internally via a (sparse) LU decomposition for explicit matrices A & M, or via an iterative solver if either A or M is a general linear operator. Alternatively, the user can supply the matrix or operator OPinv, which gives `x = OPinv * b = [A - sigma * M]^-1 * b`. For a real matrix A, shift-invert can either be done in imaginary mode or real mode, specified by the parameter OPpart ('r' or 'i'). Note that when sigma is specified, the keyword 'which' (below) refers to the shifted eigenvalues `w'[i]` where:

If A is real and OPpart == 'r' (default),

`w'[i] = 1/2 * [1/(w[i]-sigma) + 1/(w[i]-conj(sigma))].`

If A is real and OPpart == 'i',

`w'[i] = 1/2i * [1/(w[i]-sigma) - 1/(w[i]-conj(sigma))].`

If A is complex, `w'[i] = 1/(w[i]-sigma).`

v0 : ndarray, optional

Starting vector for iteration. Default: random

ncv : int, optional

The number of Lanczos vectors generated *ncv* must be greater than *k*; it is recommended that `ncv > 2*k`. Default: `min(n, max(2*k + 1, 20))`

which : str, ['LM' | 'SM' | 'LR' | 'SR' | 'LI' | 'SI'], optional
Which *k* eigenvectors and eigenvalues to find:
'LM' : largest magnitude
'SM' : smallest magnitude
'LR' : largest real part
'SR' : smallest real part
'LI' : largest imaginary part
'SI' : smallest imaginary part
When sigma != None, 'which' refers to the shifted eigenvalues w'[i] (see discussion in 'sigma', above). ARPACK is generally better at finding large values than small values. If small eigenvalues are desired, consider using shift-invert mode for better performance.

maxiter : int, optional
Maximum number of Arnoldi update iterations allowed Default: `n*10`

tol : float, optional
Relative accuracy for eigenvalues (stopping criterion) The default value of 0 implies machine precision.

return_eigenvectors : bool, optional
Return eigenvectors (True) in addition to eigenvalues

Minv : ndarray, sparse matrix or LinearOperator, optional
See notes in M, above.

OPinv : ndarray, sparse matrix or LinearOperator, optional
See notes in sigma, above.

OPpart : {'r' or 'i'}, optional
See notes in sigma, above

Returns **w** : ndarray
Array of k eigenvalues.

v : ndarray
An array of *k* eigenvectors. `v[:, i]` is the eigenvector corresponding to the eigenvalue w[i].

Raises **ArpackNoConvergence**
When the requested convergence is not obtained. The currently converged eigenvalues and eigenvectors can be found as `eigenvalues` and `eigenvectors` attributes of the exception object.

See also:

eigsh	eigenvalues and eigenvectors for symmetric matrix A
svds	singular value decomposition for a matrix A

Notes

This function is a wrapper to the ARPACK *[R327]* SNEUPD, DNEUPD, CNEUPD, ZNEUPD, functions which use the Implicitly Restarted Arnoldi Method to find the eigenvalues and eigenvectors *[R328]*.

References

[R327], *[R328]*

Examples

Find 6 eigenvectors of the identity matrix:

```
>>> import scipy.sparse as sparse
>>> id = np.eye(13)
>>> vals, vecs = sparse.linalg.eigs(id, k=6)
>>> vals
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
```

```
>>> vecs.shape
(13, 6)
```

`scipy.sparse.linalg.eigsh`(*A*, *k=6*, *M=None*, *sigma=None*, *which='LM'*, *v0=None*, *ncv=None*, *maxiter=None*, *tol=0*, *return_eigenvectors=True*, *Minv=None*, *OPinv=None*, *mode='normal'*)

Find k eigenvalues and eigenvectors of the real symmetric square matrix or complex hermitian matrix A.

Solves `A * x[i] = w[i] * x[i]`, the standard eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i].

If M is specified, solves `A * x[i] = w[i] * M * x[i]`, the generalized eigenvalue problem for w[i] eigenvalues with corresponding eigenvectors x[i]

Parameters

A : An N x N matrix, array, sparse matrix, or LinearOperator representing
the operation A * x, where A is a real symmetric matrix For buckling mode (see below) A must additionally be positive-definite

k : int, optional

The number of eigenvalues and eigenvectors desired. *k* must be smaller than N. It is not possible to compute all eigenvectors of a matrix.

Returns

w : array

Array of k eigenvalues

v : array

An array representing the *k* eigenvectors. The column `v[:, i]` is the eigenvector corresponding to the eigenvalue `w[i]`.

Other Parameters

M : An N x N matrix, array, sparse matrix, or linear operator representing
the operation M * x for the generalized eigenvalue problem

A * x = w * M * x.

M must represent a real, symmetric matrix if A is real, and must represent a complex, hermitian matrix if A is complex. For best results, the data type of M should be the same as that of A. Additionally:

If sigma is None, M is symmetric positive definite
If sigma is specified, M is symmetric positive semi-definite
In buckling mode, M is symmetric indefinite.

If sigma is None, eigsh requires an operator to compute the solution of the linear equation `M * x = b`. This is done internally via a (sparse) LU decomposition for an explicit matrix M, or via an iterative solver for a general linear operator. Alternatively, the user can supply the matrix or operator Minv, which gives `x = Minv * b = M^-1 * b`.

sigma : real

Find eigenvalues near sigma using shift-invert mode. This requires an operator to compute the solution of the linear system *[A - sigma * M] x = b*, where M is the identity matrix if unspecified. This is computed internally via a (sparse) LU decomposition for explicit matrices A & M, or via an iterative solver if either A or M is a general linear operator. Alternatively, the user can supply the matrix or operator OPinv, which gives `x = OPinv * b = [A - sigma * M]^-1 * b`. Note that when sigma is specified, the keyword 'which' refers to the shifted eigenvalues `w'[i]` where:

if mode == 'normal', `w'[i] = 1 / (w[i] - sigma)`.
if mode == 'cayley', `w'[i] = (w[i] + sigma) / (w[i] - sigma)`.
if mode == 'buckling', `w'[i] = w[i] / (w[i] - sigma)`.

(see further discussion in 'mode' below)

v0 : ndarray, optional
Starting vector for iteration. Default: random

ncv : int, optional
The number of Lanczos vectors generated ncv must be greater than k and smaller than n; it is recommended that `ncv > 2*k`. Default: `min(n, max(2*k + 1, 20))`

which : str ['LM' | 'SM' | 'LA' | 'SA' | 'BE']
If A is a complex hermitian matrix, 'BE' is invalid. Which *k* eigenvectors and eigenvalues to find:
'LM' : Largest (in magnitude) eigenvalues
'SM' : Smallest (in magnitude) eigenvalues
'LA' : Largest (algebraic) eigenvalues
'SA' : Smallest (algebraic) eigenvalues
'BE' : Half (k/2) from each end of the spectrum
When k is odd, return one more (k/2+1) from the high end. When sigma != None, 'which' refers to the shifted eigenvalues `w'[i]` (see discussion in 'sigma', above). ARPACK is generally better at finding large values than small values. If small eigenvalues are desired, consider using shift-invert mode for better performance.

maxiter : int, optional
Maximum number of Arnoldi update iterations allowed Default: `n*10`

tol : float
Relative accuracy for eigenvalues (stopping criterion). The default value of 0 implies machine precision.

Minv : N x N matrix, array, sparse matrix, or LinearOperator
See notes in M, above

OPinv : N x N matrix, array, sparse matrix, or LinearOperator
See notes in sigma, above.

return_eigenvectors : bool
Return eigenvectors (True) in addition to eigenvalues

mode : string ['normal' | 'buckling' | 'cayley']
Specify strategy to use for shift-invert mode. This argument applies only for real-valued A and sigma != None. For shift-invert mode, ARPACK internally solves the eigenvalue problem `OP * x'[i] = w'[i] * B * x'[i]` and transforms the resulting Ritz vectors x'[i] and Ritz values w'[i] into the desired eigenvectors and eigenvalues of the problem `A * x[i] = w[i] * M * x[i]`. The modes are as follows:

'normal' : OP = [A - sigma * M]^-1 * M, B = M, w'[i] = 1 / (w[i] - sigma)

'buckling' : OP = [A - sigma * M]^-1 * A, B = A, w'[i] = w[i] / (w[i] - sigma)

'cayley' : OP = [A - sigma * M]^-1 * [A + sigma * M], B = M, w'[i] = (w[i] + sigma) / (w[i] - sigma)

The choice of mode will affect which eigenvalues are selected by the keyword 'which', and can also impact the stability of convergence (see [2] for a discussion)

Raises

ArpackNoConvergence
When the requested convergence is not obtained.
The currently converged eigenvalues and eigenvectors can be found as `eigenvalues` and `eigenvectors` attributes of the exception object.

See also:

eigs eigenvalues and eigenvectors for a general (nonsymmetric) matrix A

svds singular value decomposition for a matrix A

Notes

This function is a wrapper to the ARPACK *[R329]* SSEUPD and DSEUPD functions which use the Implicitly Restarted Lanczos Method to find the eigenvalues and eigenvectors *[R330]*.

References

[R329], *[R330]*

Examples

```
>>> import scipy.sparse as sparse
>>> id = np.eye(13)
>>> vals, vecs = sparse.linalg.eigsh(id, k=6)
>>> vals
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
>>> vecs.shape
(13, 6)
```

scipy.sparse.linalg.**lobpcg**(*A, X, B=None, M=None, Y=None, tol=None, maxiter=20, largest=True, verbosityLevel=0, retLambdaHistory=False, retResidualNormsHistory=False*)

Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG)

LOBPCG is a preconditioned eigensolver for large symmetric positive definite (SPD) generalized eigenproblems.

Parameters

A : {sparse matrix, dense matrix, LinearOperator}
: The symmetric linear operator of the problem, usually a sparse matrix. Often called the "stiffness matrix".

X : array_like
: Initial approximation to the k eigenvectors. If A has shape=(n,n) then X should have shape shape=(n,k).

B : {dense matrix, sparse matrix, LinearOperator}, optional
: the right hand side operator in a generalized eigenproblem. by default, B = Identity often called the "mass matrix"

M : {dense matrix, sparse matrix, LinearOperator}, optional
: preconditioner to A; by default M = Identity M should approximate the inverse of A

Y : array_like, optional
: n-by-sizeY matrix of constraints, sizeY < n The iterations will be performed in the B-orthogonal complement of the column-space of Y. Y must be full rank.

Returns

w : array
: Array of k eigenvalues

v : array
: An array of k eigenvectors. V has the same shape as X.

Other Parameters

tol : scalar, optional
: Solver tolerance (stopping criterion) by default: tol=n*sqrt(eps)

maxiter : integer, optional
: maximum number of iterations by default: maxiter=min(n,20)

largest : bool, optional
: when True, solve for the largest eigenvalues, otherwise the smallest

verbosityLevel : integer, optional
: controls solver output. default: verbosityLevel = 0.

retLambdaHistory : boolean, optional
: whether to return eigenvalue history

retResidualNormsHistory : boolean, optional
whether to return history of residual norms

Notes

If both retLambdaHistory and retResidualNormsHistory are True, the return tuple has the following format (lambda, V, lambda history, residual norms history).

In the following `n` denotes the matrix size and `m` the number of required eigenvalues (smallest or largest).

The LOBPCG code internally solves eigenproblems of the size 3``m`` on every iteration by calling the "standard" dense eigensolver, so if `m` is not small enough compared to `n`, it does not make sense to call the LOBPCG code, but rather one should use the "standard" eigensolver, e.g. numpy or scipy function in this case. If one calls the LOBPCG algorithm for 5``m``>``n``, it will most likely break internally, so the code tries to call the standard function instead.

It is not that n should be large for the LOBPCG to work, but rather the ratio `n/m` should be large. It you call the LOBPCG code with `` m``=1 and ``n``=10, it should work, though ``n `` is small. The method is intended for extremely large `n/m`, see e.g., reference [28] in http://arxiv.org/abs/0705.2626

The convergence speed depends basically on two factors:

1. How well relatively separated the seeking eigenvalues are from the rest of the eigenvalues. One can try to vary `m` to make this better.
2. How well conditioned the problem is. This can be changed by using proper preconditioning. For example, a rod vibration test problem (under tests directory) is ill-conditioned for large `n`, so convergence will be slow, unless efficient preconditioning is used. For this specific problem, a good simple preconditioner function would be a linear solve for A, which is easy to code since A is tridiagonal.

Acknowledgements

lobpcg.py code was written by Robert Cimrman. Many thanks belong to Andrew Knyazev, the author of the algorithm, for lots of advice and support.

References

[R336], [R337], [R338]

Examples

Solve A x = lambda B x with constraints and preconditioning.

```
>>> from scipy.sparse import spdiags, issparse
>>> from scipy.sparse.linalg import lobpcg, LinearOperator
>>> n = 100
>>> vals = [np.arange(n, dtype=np.float64) + 1]
>>> A = spdiags(vals, 0, n, n)
>>> A.toarray()
array([[   1.,    0.,    0., ...,    0.,    0.,    0.],
       [   0.,    2.,    0., ...,    0.,    0.,    0.],
       [   0.,    0.,    3., ...,    0.,    0.,    0.],
       ...,
       [   0.,    0.,    0., ...,   98.,    0.,    0.],
       [   0.,    0.,    0., ...,    0.,   99.,    0.],
       [   0.,    0.,    0., ...,    0.,    0.,  100.]])
```

Constraints.

```
>>> Y = np.eye(n, 3)
```

Initial guess for eigenvectors, should have linearly independent columns. Column dimension = number of requested eigenvalues.

```
>>> X = np.random.rand(n, 3)
```

Preconditioner – inverse of A (as an abstract linear operator).

```
>>> invA = spdiags([1./vals[0]], 0, n, n)
>>> def precond( x ):
...     return invA  * x
>>> M = LinearOperator(matvec=precond, shape=(n, n), dtype=float)
```

Here, `invA` could of course have been used directly as a preconditioner. Let us then solve the problem:

```
>>> eigs, vecs = lobpcg(A, X, Y=Y, M=M, tol=1e-4, maxiter=40, largest=False)
>>> eigs
array([ 4.,  5.,  6.])
```

Note that the vectors passed in Y are the eigenvectors of the 3 smallest eigenvalues. The results returned are orthogonal to those.

Singular values problems:

svds(A[, k, ncv, tol, which, v0, maxiter, ...])	Compute the largest k singular values/vectors for a sparse matrix.

`scipy.sparse.linalg.`**`svds`**(*A*, *k=6*, *ncv=None*, *tol=0*, *which='LM'*, *v0=None*, *maxiter=None*, *return_singular_vectors=True*)

Compute the largest k singular values/vectors for a sparse matrix.

Parameters **A** : {sparse matrix, LinearOperator}

Array to compute the SVD on, of shape (M, N)

k : int, optional

Number of singular values and vectors to compute. Must be 1 <= k < min(A.shape).

ncv : int, optional

The number of Lanczos vectors generated ncv must be greater than k+1 and smaller than n; it is recommended that ncv > 2*k Default: `min(n, max(2*k + 1, 20))`

tol : float, optional

Tolerance for singular values. Zero (default) means machine precision.

which : str, ['LM' | 'SM'], optional

Which *k* singular values to find:

- 'LM' : largest singular values
- 'SM' : smallest singular values

New in version 0.12.0.

v0 : ndarray, optional

Starting vector for iteration, of length min(A.shape). Should be an (approximate) left singular vector if N > M and a right singular vector otherwise. Default: random

New in version 0.12.0.

maxiter : int, optional

Maximum number of iterations.

New in version 0.12.0.

return_singular_vectors : bool or str, optional

- True: return singular vectors (True) in addition to singular values.

New in version 0.12.0.

- "u": only return the u matrix, without computing vh (if N > M).
- "vh": only return the vh matrix, without computing u (if N <= M).

New in version 0.16.0.

Returns

u : ndarray, shape=(M, k)
Unitary matrix having left singular vectors as columns. If *return_singular_vectors* is "vh", this variable is not computed, and None is returned instead.

s : ndarray, shape=(k,)
The singular values.

vt : ndarray, shape=(k, N)
Unitary matrix having right singular vectors as rows. If *return_singular_vectors* is "u", this variable is not computed, and None is returned instead.

Notes

This is a naive implementation using ARPACK as an eigensolver on A.H * A or A * A.H, depending on which one is more efficient.

Complete or incomplete LU factorizations

`splu`(A[, permc_spec, diag_pivot_thresh, ...])	Compute the LU decomposition of a sparse, square matrix.
`spilu`(A[, drop_tol, fill_factor, drop_rule, ...])	Compute an incomplete LU decomposition for a sparse, square matrix.
`SuperLU`	LU factorization of a sparse matrix.

`scipy.sparse.linalg.`**`splu`**(*A*, *permc_spec=None*, *diag_pivot_thresh=None*, *drop_tol=None*, *relax=None*, *panel_size=None*, *options={}*)

Compute the LU decomposition of a sparse, square matrix.

Parameters

A : sparse matrix
Sparse matrix to factorize. Should be in CSR or CSC format.

permc_spec : str, optional
How to permute the columns of the matrix for sparsity preservation. (default: 'COLAMD')

- `NATURAL`: natural ordering.
- `MMD_ATA`: minimum degree ordering on the structure of A^T A.
- `MMD_AT_PLUS_A`: minimum degree ordering on the structure of A^T+A.
- `COLAMD`: approximate minimum degree column ordering

diag_pivot_thresh : float, optional
Threshold used for a diagonal entry to be an acceptable pivot. See SuperLU user's guide for details *[R349]*

drop_tol : float, optional
(deprecated) No effect.

relax : int, optional
Expert option for customizing the degree of relaxing supernodes. See SuperLU user's guide for details *[R349]*

panel_size : int, optional
Expert option for customizing the panel size. See SuperLU user's guide for details *[R349]*

options : dict, optional
Dictionary containing additional expert options to SuperLU. See SuperLU user guide *[R349]* (section 2.4 on the 'Options' argument) for more details. For example, you can specify `options=dict(Equil=False,`

IterRefine='SINGLE')) to turn equilibration off and perform a single iterative refinement.

Returns **invA** : scipy.sparse.linalg.SuperLU

Object, which has a `solve` method.

See also:

spilu incomplete LU decomposition

Notes

This function uses the SuperLU library.

References

[R349]

`scipy.sparse.linalg.`**`spilu`**(*A*, *drop_tol=None*, *fill_factor=None*, *drop_rule=None*, *permc_spec=None*, *diag_pivot_thresh=None*, *relax=None*, *panel_size=None*, *options=None*)

Compute an incomplete LU decomposition for a sparse, square matrix.

The resulting object is an approximation to the inverse of *A*.

Parameters **A** : (N, N) array_like

Sparse matrix to factorize

drop_tol : float, optional

Drop tolerance (0 <= tol <= 1) for an incomplete LU decomposition. (default: 1e-4)

fill_factor : float, optional

Specifies the fill ratio upper bound (>= 1.0) for ILU. (default: 10)

drop_rule : str, optional

Comma-separated string of drop rules to use. Available rules: `basic`, `prows`, `column`, `area`, `secondary`, `dynamic`, `interp`. (Default: `basic,area`)

See SuperLU documentation for details.

Remaining other options

Same as for *splu*

Returns **invA_approx** : scipy.sparse.linalg.SuperLU

Object, which has a `solve` method.

See also:

splu complete LU decomposition

Notes

To improve the better approximation to the inverse, you may need to increase *fill_factor* AND decrease *drop_tol*.

This function uses the SuperLU library.

class `scipy.sparse.linalg.`**`SuperLU`**

LU factorization of a sparse matrix.

Factorization is represented as:

```
Pr * A * Pc = L * U
```

To construct these *SuperLU* objects, call the *splu* and *spilu* functions.

Notes

New in version 0.14.0.

Examples

The LU decomposition can be used to solve matrix equations. Consider:

```
>>> import numpy as np
>>> from scipy.sparse import csc_matrix, linalg as sla
>>> A = csc_matrix([[1,2,0,4],[1,0,0,1],[1,0,2,1],[2,2,1,0.]])
```

This can be solved for a given right-hand side:

```
>>> lu = sla.splu(A)
>>> b = np.array([1, 2, 3, 4])
>>> x = lu.solve(b)
>>> A.dot(x)
array([ 1.,  2.,  3.,  4.])
```

The `lu` object also contains an explicit representation of the decomposition. The permutations are represented as mappings of indices:

```
>>> lu.perm_r
array([0, 2, 1, 3], dtype=int32)
>>> lu.perm_c
array([2, 0, 1, 3], dtype=int32)
```

The L and U factors are sparse matrices in CSC format:

```
>>> lu.L.A
array([[ 1. ,  0. ,  0. ,  0. ],
       [ 0. ,  1. ,  0. ,  0. ],
       [ 0. ,  0. ,  1. ,  0. ],
       [ 1. ,  0.5,  0.5,  1. ]])
>>> lu.U.A
array([[ 2.,  0.,  1.,  4.],
       [ 0.,  2.,  1.,  1.],
       [ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  0., -5.]])
```

The permutation matrices can be constructed:

```
>>> Pr = csc_matrix((4, 4))
>>> Pr[lu.perm_r, np.arange(4)] = 1
>>> Pc = csc_matrix((4, 4))
>>> Pc[np.arange(4), lu.perm_c] = 1
```

We can reassemble the original matrix:

```
>>> (Pr.T * (lu.L * lu.U) * Pc.T).A
array([[ 1.,  2.,  0.,  4.],
       [ 1.,  0.,  0.,  1.],
       [ 1.,  0.,  2.,  1.],
       [ 2.,  2.,  1.,  0.]])
```

Attributes

shape	Shape of the original matrix as a tuple of ints.
nnz	Number of nonzero elements in the matrix.

Continued on next page

Table 5.202 – continued from previous page

perm_c	Permutation Pc represented as an array of indices.
perm_r	Permutation Pr represented as an array of indices.
L	Lower triangular factor with unit diagonal as a *scipy.sparse.csc_matrix*.
U	Upper triangular factor as a *scipy.sparse.csc_matrix*.

SuperLU.**shape**
Shape of the original matrix as a tuple of ints.

SuperLU.**nnz**
Number of nonzero elements in the matrix.

SuperLU.**perm_c**
Permutation Pc represented as an array of indices.

The column permutation matrix can be reconstructed via:

```
>>> Pc = np.zeros((n, n))
>>> Pc[np.arange(n), perm_c] = 1
```

SuperLU.**perm_r**
Permutation Pr represented as an array of indices.

The row permutation matrix can be reconstructed via:

```
>>> Pr = np.zeros((n, n))
>>> Pr[perm_r, np.arange(n)] = 1
```

SuperLU.**L**
Lower triangular factor with unit diagonal as a *scipy.sparse.csc_matrix*.

New in version 0.14.0.

SuperLU.**U**
Upper triangular factor as a *scipy.sparse.csc_matrix*.

New in version 0.14.0.

Methods

solve(rhs[, trans])	Solves linear system of equations with one or several right-hand sides.

SuperLU.**solve**(*rhs*[, *trans*])
Solves linear system of equations with one or several right-hand sides.

Parameters **rhs** : ndarray, shape (n,) or (n, k)
Right hand side(s) of equation

trans : {‘N’, ‘T’, ‘H’}, optional
Type of system to solve:

```
'N':   A   * x == rhs  (default)
'T':   A^T * x == rhs
'H':   A^H * x == rhs
```

i.e., normal, transposed, and hermitian conjugate.

Returns **x** : ndarray, shape `rhs.shape`
Solution vector(s)

5.22.6 Exceptions

ArpackNoConvergence(msg, eigenvalues, ...)	ARPACK iteration did not converge
ArpackError(info[, infodict])	ARPACK error

exception `scipy.sparse.linalg.`**`ArpackNoConvergence`**(*msg*, *eigenvalues*, *eigenvectors*)
ARPACK iteration did not converge

Attributes

eigenvalues	(ndarray) Partial result. Converged eigenvalues.
eigenvectors	(ndarray) Partial result. Converged eigenvectors.

exception `scipy.sparse.linalg.ArpackError`(*info*, *infodict={'c': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size of the current Arnoldi factorization. The user is advised to check that enough workspace and array storage has been allocated.', -13: "NEV and WHICH = 'BE' are incompatible.", -12: 'IPARAM(1) must be equal to 0 or 1.', -1: 'N must be positive.', -10: 'IPARAM(7) must be 1, 2, 3.', -9: 'Starting vector is zero.', -8: 'Error return from LAPACK eigenvalue calculation;', -7: 'Length of private work array WORKL is not sufficient.', -6: "BMAT must be one of 'I' or 'G'.", -5: " WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI"', -4: 'The maximum number of Arnoldi update iterations allowed must be greater than zero.', -3: 'NCV-NEV >= 2 and less than or equal to N.', -2: 'NEV must be positive.', -11: "IPARAM(7) = 1 and BMAT = 'G' are incompatible."}, 's': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size of the current Arnoldi factorization. The user is advised to check that enough workspace and array storage has been allocated.', -13: "NEV and WHICH = 'BE' are incompatible.", -12: 'IPARAM(1) must be equal to 0 or 1.', -2: 'NEV must be positive.', -10: 'IPARAM(7) must be 1, 2, 3, 4.', -9: 'Starting vector is zero.', -8: 'Error return from LAPACK eigenvalue calculation;', -7: 'Length of private work array WORKL is not sufficient.', -6: "BMAT must be one of 'I' or 'G'.", -5: " WHICH must be one of 'LM', 'SM', 'LR', 'SR', 'LI', 'SI"', -4: 'The maximum number of Arnoldi update iterations allowed must be greater than zero.', -3: 'NCV-NEV >= 2 and less than or equal to N.', -1: 'N must be positive.', -11: "IPARAM(7) = 1 and BMAT = 'G' are incompatible."}, 'z': {0: 'Normal exit.', 1: 'Maximum number of iterations taken. All possible eigenvalues of OP has been found. IPARAM(5) returns the number of wanted converged Ritz values.', 2: 'No longer an informational error. Deprecated starting with release 2 of ARPACK.', 3: 'No shifts could be applied during a cycle of the Implicitly restarted Arnoldi iteration. One possibility is to increase the size of NCV relative to NEV. ', -9999: 'Could not build an Arnoldi factorization. IPARAM(5) returns the size*

ARPACK error

5.23 Compressed Sparse Graph Routines (`scipy.sparse.csgraph`)

Fast graph algorithms based on sparse matrix representations.

5.23.1 Contents

connected_components(csgraph[, directed, ...])	Analyze the connected components of a sparse graph
laplacian(csgraph[, normed, return_diag, ...])	Return the Laplacian matrix of a directed graph.
shortest_path(csgraph[, method, directed, ...])	Perform a shortest-path graph search on a positive directed or undirected graph.
dijkstra(csgraph[, directed, indices, ...])	Dijkstra algorithm using Fibonacci Heaps
floyd_warshall(csgraph[, directed, ...])	Compute the shortest path lengths using the Floyd-Warshall algorithm
bellman_ford(csgraph[, directed, indices, ...])	Compute the shortest path lengths using the Bellman-Ford algorithm.
johnson(csgraph[, directed, indices, ...])	Compute the shortest path lengths using Johnson's algorithm.
breadth_first_order(csgraph, i_start[, ...])	Return a breadth-first ordering starting with specified node.
depth_first_order(csgraph, i_start[, ...])	Return a depth-first ordering starting with specified node.
breadth_first_tree(csgraph, i_start[, directed])	Return the tree generated by a breadth-first search
depth_first_tree(csgraph, i_start[, directed])	Return a tree generated by a depth-first search.
minimum_spanning_tree(csgraph[, overwrite])	Return a minimum spanning tree of an undirected graph
reverse_cuthill_mckee(graph[, symmetric_mode])	Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering.
maximum_bipartite_matching(graph[, perm_type])	Returns an array of row or column permutations that makes the diagonal of a nonsingular square CSC sparse matrix zero free.
structural_rank(graph)	Compute the structural rank of a graph (matrix) with a given sparsity pattern.
NegativeCycleError	

scipy.sparse.csgraph.**connected_components**(*csgraph*, *directed=True*, *connection='weak'*, *return_labels=True*)

Analyze the connected components of a sparse graph

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

connection : str, optional

['weak'|'strong']. For directed graphs, the type of connection to use. Nodes i and j are strongly connected if a path exists both from i to j and from j to

i. Nodes i and j are weakly connected if only one of these paths exists. If directed == False, this keyword is not referenced.

return_labels : bool, optional
If True (default), then return the labels for each of the connected components.

Returns n_components: int
The number of connected components.

labels: ndarray
The length-N array of labels of the connected components.

References

[R278]

`scipy.sparse.csgraph.laplacian`(*csgraph*, *normed=False*, *return_diag=False*, *use_out_degree=False*)

Return the Laplacian matrix of a directed graph.

Parameters **csgraph** : array_like or sparse matrix, 2 dimensions
compressed-sparse graph, with shape (N, N).

normed : bool, optional
If True, then compute normalized Laplacian.

return_diag : bool, optional
If True, then also return an array related to vertex degrees.

use_out_degree : bool, optional
If True, then use out-degree instead of in-degree. This distinction matters only if the graph is asymmetric. Default: False.

Returns **lap** : ndarray or sparse matrix
The N x N laplacian matrix of csgraph. It will be a numpy array (dense) if the input was dense, or a sparse matrix otherwise.

diag : ndarray, optional
The length-N diagonal of the Laplacian matrix. For the normalized Laplacian, this is the array of square roots of vertex degrees or 1 if the degree is zero.

Notes

The Laplacian matrix of a graph is sometimes referred to as the "Kirchoff matrix" or the "admittance matrix", and is useful in many parts of spectral graph theory. In particular, the eigen-decomposition of the laplacian matrix can give insight into many properties of the graph.

Examples

```
>>> from scipy.sparse import csgraph
>>> G = np.arange(5) * np.arange(5)[:, np.newaxis]
>>> G
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12],
       [ 0,  4,  8, 12, 16]])
>>> csgraph.laplacian(G, normed=False)
array([[  0,   0,   0,   0,   0],
       [  0,   9,  -2,  -3,  -4],
       [  0,  -2,  16,  -6,  -8],
       [  0,  -3,  -6,  21, -12],
       [  0,  -4,  -8, -12,  24]])
```

scipy.sparse.csgraph.**shortest_path**(*csgraph*, *method='auto'*, *directed=True*, *return_predecessors=False*, *unweighted=False*, *overwrite=False*, *indices=None*)

Perform a shortest-path graph search on a positive directed or undirected graph.

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

method : string ['auto'|'FW'|'D'], optional
Algorithm to use for shortest paths. Options are:

'auto' – (default) select the best among 'FW', 'D', 'BF', or 'J'
based on the input data.

'FW' – Floyd-Warshall algorithm. Computational cost is
approximately `O[N^3]`. The input csgraph will be converted to a dense representation.

'D' – Dijkstra's algorithm with Fibonacci heaps. Computational
cost is approximately `O[N(N*k + N*log(N))]`, where `k` is the average number of connected edges per node. The input csgraph will be converted to a csr representation.

'BF' – Bellman-Ford algorithm. This algorithm can be used when
weights are negative. If a negative cycle is encountered, an error will be raised. Computational cost is approximately `O[N(N^2 k)]`, where `k` is the average number of connected edges per node. The input csgraph will be converted to a csr representation.

'J' – Johnson's algorithm. Like the Bellman-Ford algorithm,
Johnson's algorithm is designed for use when the weights are negative. It combines the Bellman-Ford algorithm with Dijkstra's algorithm for faster computation.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

overwrite : bool, optional
If True, overwrite csgraph with the result. This applies only if method == 'FW' and csgraph is a dense, c-ordered array with dtype=float64.

indices : array_like or int, optional
If specified, only compute the paths for the points at the given indices. Incompatible with method == 'FW'.

Returns

dist_matrix : ndarray

The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray

Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises **NegativeCycleError:**

if there are negative cycles in the graph

Notes

As currently implemented, Dijkstra's algorithm and Johnson's algorithm do not work for graphs with direction-dependent distances when directed == False. i.e., if csgraph[i,j] and csgraph[j,i] are non-equal edges, method='D' may yield an incorrect result.

`scipy.sparse.csgraph.`**`dijkstra`**(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*, *limit=np.inf*)

Dijkstra algorithm using Fibonacci Heaps

New in version 0.11.0.

Parameters **csgraph** : array, matrix, or sparse matrix, 2 dimensions

The N x N array of non-negative distances representing the input graph.

directed : bool, optional

If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional

if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional

If True, return the size (N, N) predecesor matrix

unweighted : bool, optional

If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

limit : float, optional

The maximum distance to calculate, must be >= 0. Using a smaller limit will decrease computation time by aborting calculations between pairs that are separated by a distance > limit. For such pairs, the distance will be equal to np.inf (i.e., not connected).

New in version 0.14.0.

Returns **dist_matrix** : ndarray

The matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray

Returned only if return_predecessors == True. The matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Notes

As currently implemented, Dijkstra's algorithm does not work for graphs with direction-dependent distances when directed == False. i.e., if csgraph[i,j] and csgraph[j,i] are not equal and both are nonzero, setting directed=False will not yield the correct result.

Also, this routine does not work for graphs with negative distances. Negative distances can lead to infinite cycles that must be handled by specialized algorithms such as Bellman-Ford's algorithm or Johnson's algorithm.

scipy.sparse.csgraph.**floyd_warshall**(*csgraph*, *directed=True*, *return_predecessors=False*, *unweighted=False*, *overwrite=False*)

Compute the shortest path lengths using the Floyd-Warshall algorithm

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

overwrite : bool, optional
If True, overwrite csgraph with the result. This applies only if csgraph is a dense, c-ordered array with dtype=float64.

Returns

dist_matrix : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises

NegativeCycleError:
if there are negative cycles in the graph

scipy.sparse.csgraph.**bellman_ford**(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*)

Compute the shortest path lengths using the Bellman-Ford algorithm.

The Bellman-ford algorithm can robustly deal with graphs with negative weights. If a negative cycle is detected, an error is raised. For graphs without negative edge weights, dijkstra's algorithm may be faster.

New in version 0.11.0.

Parameters

csgraph : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional
if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns **dist_matrix** : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises **NegativeCycleError:**
if there are negative cycles in the graph

Notes

This routine is specially designed for graphs with negative edge weights. If all edge weights are positive, then Dijkstra's algorithm is a better choice.

`scipy.sparse.csgraph.`**`johnson`**(*csgraph*, *directed=True*, *indices=None*, *return_predecessors=False*, *unweighted=False*)

Compute the shortest path lengths using Johnson's algorithm.

Johnson's algorithm combines the Bellman-Ford algorithm and Dijkstra's algorithm to quickly find shortest paths in a way that is robust to the presence of negative cycles. If a negative cycle is detected, an error is raised. For graphs without negative edge weights, dijkstra() may be faster.

New in version 0.11.0.

Parameters **csgraph** : array, matrix, or sparse matrix, 2 dimensions
The N x N array of distances representing the input graph.

directed : bool, optional
If True (default), then find the shortest path on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i]

indices : array_like or int, optional
if specified, only compute the paths for the points at the given indices.

return_predecessors : bool, optional
If True, return the size (N, N) predecesor matrix

unweighted : bool, optional
If True, then find unweighted distances. That is, rather than finding the path between each point such that the sum of weights is minimized, find the path such that the number of edges is minimized.

Returns **dist_matrix** : ndarray
The N x N matrix of distances between graph nodes. dist_matrix[i,j] gives the shortest distance from point i to point j along the graph.

predecessors : ndarray
Returned only if return_predecessors == True. The N x N matrix of predecessors, which can be used to reconstruct the shortest paths. Row i of the predecessor matrix contains information on the shortest paths from point

i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

Raises **NegativeCycleError:**

if there are negative cycles in the graph

Notes

This routine is specially designed for graphs with negative edge weights. If all edge weights are positive, then Dijkstra's algorithm is a better choice.

`scipy.sparse.csgraph.`**`breadth_first_order`**(*csgraph*, *i_start*, *directed=True*, *return_predecessors=True*)

Return a breadth-first ordering starting with specified node.

Note that a breadth-first order is not unique, but the tree which it generates is unique.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int

The index of starting node.

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

return_predecessors : bool, optional

If True (default), then return the predecesor array (see below).

Returns **node_array** : ndarray, one dimension

The breadth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.

predecessors : ndarray, one dimension

Returned only if return_predecessors is True. The length-N list of predecessors of each node in a breadth-first tree. If node i is in the tree, then its parent is given by predecessors[i]. If node i is not in the tree (and for the parent node) then predecessors[i] = -9999.

`scipy.sparse.csgraph.`**`depth_first_order`**(*csgraph*, *i_start*, *directed=True*, *return_predecessors=True*)

Return a depth-first ordering starting with specified node.

Note that a depth-first order is not unique. Furthermore, for graphs with cycles, the tree generated by a depth-first search is not unique either.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int

The index of starting node.

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

return_predecessors : bool, optional

If True (default), then return the predecesor array (see below).

Returns **node_array** : ndarray, one dimension

The breadth-first list of nodes, starting with specified node. The length of node_array is the number of nodes reachable from the specified node.

predecessors : ndarray, one dimension

Returned only if return_predecessors is True. The length-N list of predecessors of each node in a breadth-first tree. If node i is in the tree, then its parent is given by predecessors[i]. If node i is not in the tree (and for the parent node) then predecessors[i] = -9999.

`scipy.sparse.csgraph.`**`breadth_first_tree`**(*csgraph*, *i_start*, *directed=True*)

Return the tree generated by a breadth-first search

Note that a breadth-first tree from a specified node is unique.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int

The index of starting node.

directed : bool, optional

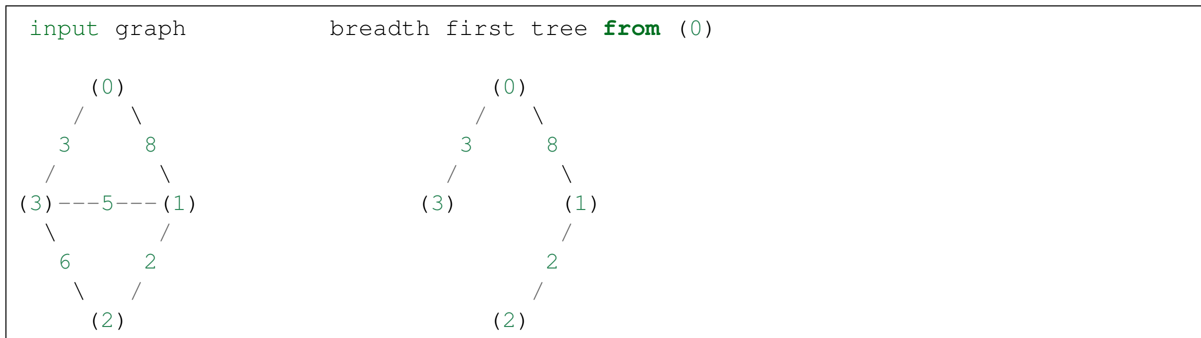
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns **cstree** : csr matrix

The N x N directed compressed-sparse representation of the breadth- first tree drawn from csgraph, starting at the specified node.

Examples

The following example shows the computation of a depth-first tree over a simple four-component graph, starting at node 0:



In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import breadth_first_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = breadth_first_tree(X, 0, directed=False)
>>> Tcsr.toarray().astype(int)
array([[0, 8, 0, 3],
```

```
        [0, 0, 2, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

Note that the resulting graph is a Directed Acyclic Graph which spans the graph. A breadth-first tree from a given node is unique.

`scipy.sparse.csgraph.depth_first_tree`(*csgraph*, *i_start*, *directed=True*)

Return a tree generated by a depth-first search.

Note that a tree generated by a depth-first search is not unique: it depends on the order that the children of each node are searched.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N matrix representing the compressed sparse graph. The input csgraph will be converted to csr format for the calculation.

i_start : int

The index of starting node.

directed : bool, optional

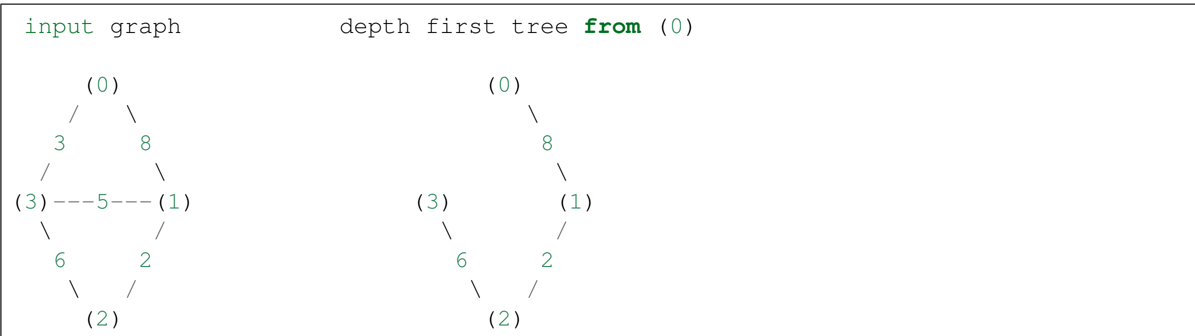
If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns **cstree** : csr matrix

The N x N directed compressed-sparse representation of the depth- first tree drawn from csgraph, starting at the specified node.

Examples

The following example shows the computation of a depth-first tree over a simple four-component graph, starting at node 0:



```
 input graph          depth first tree from (0)

     (0)                        (0)
    /   \                          \
   3     8                          8
  /       \                          \
(3)---5---(1)               (3)       (1)
  \       /                   \       /
   6     2                     6     2
    \   /                       \   /
     (2)                         (2)
```

In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import depth_first_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = depth_first_tree(X, 0, directed=False)
>>> Tcsr.toarray().astype(int)
array([[0, 8, 0, 0],
       [0, 0, 2, 0],
```

```
        [0, 0, 0, 6],
        [0, 0, 0, 0]])
```

Note that the resulting graph is a Directed Acyclic Graph which spans the graph. Unlike a breadth-first tree, a depth-first tree of a given graph is not unique if the graph contains cycles. If the above solution had begun with the edge connecting nodes 0 and 3, the result would have been different.

`scipy.sparse.csgraph.`**`minimum_spanning_tree`**(*csgraph*, *overwrite=False*)

Return a minimum spanning tree of an undirected graph

A minimum spanning tree is a graph consisting of the subset of edges which together connect all connected nodes, while minimizing the total sum of weights on the edges. This is computed using the Kruskal algorithm.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix, 2 dimensions

The N x N matrix representing an undirected graph over N nodes (see notes below).

overwrite : bool, optional

if true, then parts of the input graph will be overwritten for efficiency.

Returns **span_tree** : csr matrix

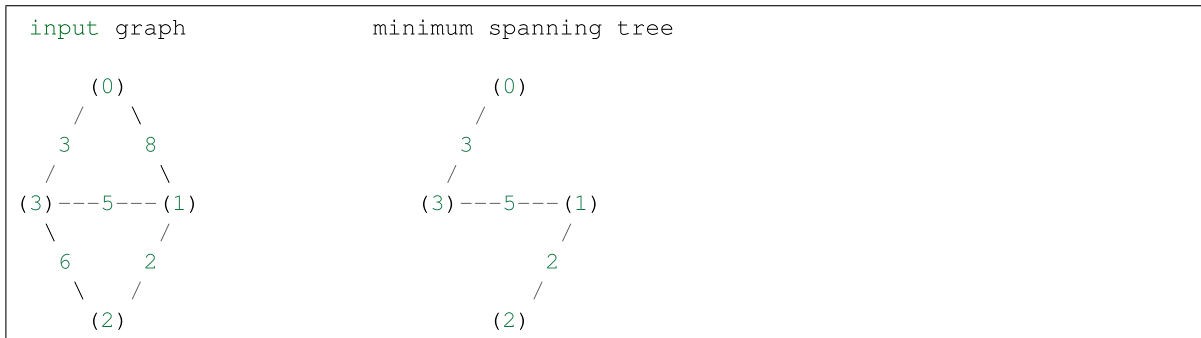
The N x N compressed-sparse representation of the undirected minimum spanning tree over the input (see notes below).

Notes

This routine uses undirected graphs as input and output. That is, if graph[i, j] and graph[j, i] are both zero, then nodes i and j do not have an edge connecting them. If either is nonzero, then the two are connected by the minimum nonzero value of the two.

Examples

The following example shows the computation of a minimum spanning tree over a simple four-component graph:



```
 input graph             minimum spanning tree

     (0)                         (0)
    /   \                       /
   3     8                     3
  /       \                   /
(3)---5---(1)               (3)---5---(1)
  \       /                           /
   6     2                           2
    \   /                           /
     (2)                         (2)
```

It is easy to see from inspection that the minimum spanning tree involves removing the edges with weights 8 and 6. In compressed sparse representation, the solution looks like this:

```
>>> from scipy.sparse import csr_matrix
>>> from scipy.sparse.csgraph import minimum_spanning_tree
>>> X = csr_matrix([[0, 8, 0, 3],
...                 [0, 0, 2, 5],
...                 [0, 0, 0, 6],
...                 [0, 0, 0, 0]])
>>> Tcsr = minimum_spanning_tree(X)
>>> Tcsr.toarray().astype(int)
array([[0, 0, 0, 3],
```

```
        [0, 0, 2, 5],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

`scipy.sparse.csgraph.`**`reverse_cuthill_mckee`**(*graph*, *symmetric_mode=False*)

Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering.

It is assumed by default, `symmetric_mode=False`, that the input matrix is not symmetric and works on the matrix `A+A.T`. If you are guaranteed that the matrix is symmetric in structure (values of matrix elements do not matter) then set `symmetric_mode=True`.

Parameters **graph** : sparse matrix

Input sparse in CSC or CSR sparse matrix format.

symmetric_mode : bool, optional

Is input matrix guaranteed to be symmetric.

Returns **perm** : ndarray

Array of permuted row and column indices.

Notes

New in version 0.15.0.

References

E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", ACM '69 Proceedings of the 1969 24th national conference, (1969).

`scipy.sparse.csgraph.`**`maximum_bipartite_matching`**(*graph*, *perm_type='row'*)

Returns an array of row or column permutations that makes the diagonal of a nonsingular square CSC sparse matrix zero free.

Such a permutation is always possible provided that the matrix is nonsingular. This function looks at the structure of the matrix only. The input matrix will be converted to CSC matrix format if necessary.

Parameters **graph** : sparse matrix

Input sparse in CSC format

perm_type : str, {'row', 'column'}

Type of permutation to generate.

Returns **perm** : ndarray

Array of row or column permutations.

Notes

This function relies on a maximum cardinality bipartite matching algorithm based on a breadth-first search (BFS) of the underlying graph.

New in version 0.15.0.

References

I. S. Duff, K. Kaya, and B. Ucar, "Design, Implementation, and Analysis of Maximum Transversal Algorithms", ACM Trans. Math. Softw. 38, no. 2, (2011).

`scipy.sparse.csgraph.`**`structural_rank`**(*graph*)

Compute the structural rank of a graph (matrix) with a given sparsity pattern.

The structural rank of a matrix is the number of entries in the maximum transversal of the corresponding bipartite graph, and is an upper bound on the numerical rank of the matrix. A graph has full structural rank if it is possible to permute the elements to make the diagonal zero-free.

Parameters **graph** : sparse matrix

Input sparse matrix.

Returns **rank** : int

The structural rank of the sparse graph.

New in version 0.19.0.

References

[R279], *[R280]*

exception `scipy.sparse.csgraph.`**`NegativeCycleError`**

construct_dist_matrix(graph, predecessors[, ...])	Construct distance matrix from a predecessor matrix
csgraph_from_dense(graph[, null_value, ...])	Construct a CSR-format sparse graph from a dense matrix.
csgraph_from_masked(graph)	Construct a CSR-format graph from a masked array.
csgraph_masked_from_dense(graph[, ...])	Construct a masked array graph representation from a dense matrix.
csgraph_to_dense(csgraph[, null_value])	Convert a sparse graph representation to a dense representation
csgraph_to_masked(csgraph)	Convert a sparse graph representation to a masked array representation
reconstruct_path(csgraph, predecessors[, ...])	Construct a tree from a graph and a predecessor list.

`scipy.sparse.csgraph.`**`construct_dist_matrix`**(*graph*, *predecessors*, *directed=True*, *null_value=np.inf*)

Construct distance matrix from a predecessor matrix

New in version 0.11.0.

Parameters **graph** : array_like or sparse

The N x N matrix representation of a directed or undirected graph. If dense, then non-edges are indicated by zeros or infinities.

predecessors : array_like

The N x N matrix of predecessors of each node (see Notes below).

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then operate on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

null_value : bool, optional

value to use for distances between unconnected nodes. Default is np.inf

Returns **dist_matrix** : ndarray

The N x N matrix of distances between nodes along the path specified by the predecessor matrix. If no path exists, the distance is zero.

Notes

The predecessor matrix is of the form returned by `graph_shortest_path`. Row i of the predecessor matrix contains information on the shortest paths from point i: each entry predecessors[i, j] gives the index of the previous node in the path from point i to point j. If no path exists between point i and j, then predecessors[i, j] = -9999

`scipy.sparse.csgraph.`**`csgraph_from_dense`**(*graph*, *null_value=0*, *nan_null=True*, *infinity_null=True*)

Construct a CSR-format sparse graph from a dense matrix.

New in version 0.11.0.

Parameters **graph** : array_like

Input graph. Shape should be (n_nodes, n_nodes).

null_value : float or None (optional)
Value that denotes non-edges in the graph. Default is zero.

infinity_null : bool
If True (default), then infinite entries (both positive and negative) are treated as null edges.

nan_null : bool
If True (default), then NaN entries are treated as non-edges

Returns **csgraph** : csr_matrix
Compressed sparse representation of graph,

`scipy.sparse.csgraph.csgraph_from_masked(graph)`

Construct a CSR-format graph from a masked array.

New in version 0.11.0.

Parameters **graph** : MaskedArray
Input graph. Shape should be (n_nodes, n_nodes).

Returns **csgraph** : csr_matrix
Compressed sparse representation of graph,

`scipy.sparse.csgraph.csgraph_masked_from_dense(graph, null_value=0, nan_null=True, infinity_null=True, copy=True)`

Construct a masked array graph representation from a dense matrix.

New in version 0.11.0.

Parameters **graph** : array_like
Input graph. Shape should be (n_nodes, n_nodes).

null_value : float or None (optional)
Value that denotes non-edges in the graph. Default is zero.

infinity_null : bool
If True (default), then infinite entries (both positive and negative) are treated as null edges.

nan_null : bool
If True (default), then NaN entries are treated as non-edges

Returns **csgraph** : MaskedArray
masked array representation of graph

`scipy.sparse.csgraph.csgraph_to_dense(csgraph, null_value=0)`

Convert a sparse graph representation to a dense representation

New in version 0.11.0.

Parameters **csgraph** : csr_matrix, csc_matrix, or lil_matrix
Sparse representation of a graph.

null_value : float, optional
The value used to indicate null edges in the dense representation. Default is 0.

Returns **graph** : ndarray
The dense representation of the sparse graph.

Notes

For normal sparse graph representations, calling csgraph_to_dense with null_value=0 produces an equivalent result to using dense format conversions in the main sparse package. When the sparse representations have repeated values, however, the results will differ. The tools in scipy.sparse will add repeating values to obtain a final value. This function will select the minimum among repeating values to obtain a final value. For example, here we'll create a two-node directed sparse graph with multiple edges from node 0 to node 1, of weights 2 and 3. This illustrates the difference in behavior:

```
>>> from scipy.sparse import csr_matrix, csgraph
>>> data = np.array([2, 3])
>>> indices = np.array([1, 1])
>>> indptr = np.array([0, 2, 2])
>>> M = csr_matrix((data, indices, indptr), shape=(2, 2))
>>> M.toarray()
array([[0, 5],
       [0, 0]])
>>> csgraph.csgraph_to_dense(M)
array([[0., 2.],
       [0., 0.]])
```

The reason for this difference is to allow a compressed sparse graph to represent multiple edges between any two nodes. As most sparse graph algorithms are concerned with the single lowest-cost edge between any two nodes, the default scipy.sparse behavior of summming multiple weights does not make sense in this context.

The other reason for using this routine is to allow for graphs with zero-weight edges. Let's look at the example of a two-node directed graph, connected by an edge of weight zero:

```
>>> from scipy.sparse import csr_matrix, csgraph
>>> data = np.array([0.0])
>>> indices = np.array([1])
>>> indptr = np.array([0, 1, 1])
>>> M = csr_matrix((data, indices, indptr), shape=(2, 2))
>>> M.toarray()
array([[0, 0],
       [0, 0]])
>>> csgraph.csgraph_to_dense(M, np.inf)
array([[ inf,   0.],
       [ inf,  inf]])
```

In the first case, the zero-weight edge gets lost in the dense representation. In the second case, we can choose a different null value and see the true form of the graph.

`scipy.sparse.csgraph.`**`csgraph_to_masked`**(*csgraph*)

Convert a sparse graph representation to a masked array representation

New in version 0.11.0.

Parameters **csgraph** : csr_matrix, csc_matrix, or lil_matrix

Sparse representation of a graph.

Returns **graph** : MaskedArray

The masked dense representation of the sparse graph.

`scipy.sparse.csgraph.`**`reconstruct_path`**(*csgraph*, *predecessors*, *directed=True*)

Construct a tree from a graph and a predecessor list.

New in version 0.11.0.

Parameters **csgraph** : array_like or sparse matrix

The N x N matrix representing the directed or undirected graph from which the predecessors are drawn.

predecessors : array_like, one dimension

The length-N array of indices of predecessors for the tree. The index of the parent of node i is given by predecessors[i].

directed : bool, optional

If True (default), then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then operate on an undirected

graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].

Returns **cstree** : csr matrix

The N x N directed compressed-sparse representation of the tree drawn from csgraph which is encoded by the predecessor list.

5.23.2 Graph Representations

This module uses graphs which are stored in a matrix format. A graph with N nodes can be represented by an (N x N) adjacency matrix G. If there is a connection from node i to node j, then G[i, j] = w, where w is the weight of the connection. For nodes i and j which are not connected, the value depends on the representation:

- for dense array representations, non-edges are represented by G[i, j] = 0, infinity, or NaN.
- for dense masked representations (of type np.ma.MaskedArray), non-edges are represented by masked values. This can be useful when graphs with zero-weight edges are desired.
- for sparse array representations, non-edges are represented by non-entries in the matrix. This sort of sparse representation also allows for edges with zero weights.

As a concrete example, imagine that you would like to represent the following undirected graph:

This graph has three nodes, where node 0 and 1 are connected by an edge of weight 2, and nodes 0 and 2 are connected by an edge of weight 1. We can construct the dense, masked, and sparse representations as follows, keeping in mind that an undirected graph is represented by a symmetric matrix:

```
>>> G_dense = np.array([[0, 2, 1],
...                     [2, 0, 0],
...                     [1, 0, 0]])
>>> G_masked = np.ma.masked_values(G_dense, 0)
>>> from scipy.sparse import csr_matrix
>>> G_sparse = csr_matrix(G_dense)
```

This becomes more difficult when zero edges are significant. For example, consider the situation when we slightly modify the above graph:

This is identical to the previous graph, except nodes 0 and 2 are connected by an edge of zero weight. In this case, the dense representation above leads to ambiguities: how can non-edges be represented if zero is a meaningful value? In this case, either a masked or sparse representation must be used to eliminate the ambiguity:

```
>>> G2_data = np.array([[np.inf, 2,      0     ],
...                     [2,      np.inf, np.inf],
```

```
...                      [0,      np.inf, np.inf]])
>>> G2_masked = np.ma.masked_invalid(G2_data)
>>> from scipy.sparse.csgraph import csgraph_from_dense
>>> # G2_sparse = csr_matrix(G2_data) would give the wrong result
>>> G2_sparse = csgraph_from_dense(G2_data, null_value=np.inf)
>>> G2_sparse.data
array([ 2.,  0.,  2.,  0.])
```

Here we have used a utility routine from the csgraph submodule in order to convert the dense representation to a sparse representation which can be understood by the algorithms in submodule. By viewing the data array, we can see that the zero values are explicitly encoded in the graph.

Directed vs. Undirected

Matrices may represent either directed or undirected graphs. This is specified throughout the csgraph module by a boolean keyword. Graphs are assumed to be directed by default. In a directed graph, traversal from node i to node j can be accomplished over the edge G[i, j], but not the edge G[j, i]. In a non-directed graph, traversal from node i to node j can be accomplished over either G[i, j] or G[j, i]. If both edges are not null, and the two have unequal weights, then the smaller of the two is used. Note that a symmetric matrix will represent an undirected graph, regardless of whether the 'directed' keyword is set to True or False. In this case, using `directed=True` generally leads to more efficient computation.

The routines in this module accept as input either scipy.sparse representations (csr, csc, or lil format), masked representations, or dense representations with non-edges indicated by zeros, infinities, and NaN entries.

5.24 Spatial algorithms and data structures (`scipy.spatial`)

5.24.1 Nearest-neighbor Queries

KDTree(data[, leafsize])	kd-tree for quick nearest-neighbor lookup
cKDTree	kd-tree for quick nearest-neighbor lookup
distance	
Rectangle(maxes, mins)	Hyperrectangle class.

class `scipy.spatial.`**`KDTree`**(*data*, *leafsize=10*)

kd-tree for quick nearest-neighbor lookup

This class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors of any point.

Parameters

data : (N,K) array_like

The data points to be indexed. This array is not copied, and so modifying this data will result in bogus results.

leafsize : int, optional

The number of points at which the algorithm switches over to brute-force. Has to be positive.

Raises

RuntimeError

The maximum recursion limit can be exceeded for large data sets. If this happens, either increase the value for the *leafsize* parameter or increase the recursion limit by:

```
>>> import sys
>>> sys.setrecursionlimit(10000)
```

See also:

cKDTree Implementation of *KDTree* in Cython

Notes

The algorithm used is described in Maneewongvatana and Mount 1999. The general idea is that the kd-tree is a binary tree, each of whose nodes represents an axis-aligned hyperrectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value.

During construction, the axis and splitting point are chosen by the "sliding midpoint" rule, which ensures that the cells do not all become long and thin.

The tree can be queried for the r closest neighbors of any given point (optionally returning only those within some maximum distance of the point). It can also be queried, with a substantial gain in efficiency, for the r approximate closest neighbors.

For large dimensions (20 is already large) do not expect this to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.

The tree also supports all-neighbors queries, both with arrays of points and with other kd-trees. These do use a reasonably efficient algorithm, but the kd-tree is not necessarily the best data structure for this sort of calculation.

Methods

count_neighbors(other, r[, p])	Count how many nearby pairs can be formed.
innernode(split_dim, split, less, greater)	
leafnode(idx)	
node	
query(x[, k, eps, p, distance_upper_bound])	Query the kd-tree for nearest neighbors
query_ball_point(x, r[, p, eps])	Find all points within distance r of point(s) x.
query_ball_tree(other, r[, p, eps])	Find all pairs of points whose distance is at most r
query_pairs(r[, p, eps])	Find all pairs of points within a distance.
sparse_distance_matrix(other, max_distance)	Compute a sparse distance matrix

KDTree.**count_neighbors**(*other*, *r*, *p=2.0*)

Count how many nearby pairs can be formed.

Count the number of pairs (x1,x2) can be formed, with x1 drawn from self and x2 drawn from *other*, and where `distance(x1, x2, p) <= r`. This is the "two-point correlation" described in Gray and Moore 2000, "N-body problems in statistical learning", and the code here is based on their algorithm.

Parameters

other : KDTree instance
  The other tree to draw points from.

r : float or one-dimensional array of floats
  The radius to produce a count for. Multiple radii are searched with a single tree traversal.

p : float, 1<=p<=infinity, optional
  Which Minkowski p-norm to use

Returns

result : int or 1-D array of ints
  The number of pairs. Note that this is internally stored in a numpy int, and so may overflow if very large (2e9).

`KDTree.query(x, k=1, eps=0, p=2, distance_upper_bound=inf)`
Query the kd-tree for nearest neighbors

Parameters

x : array_like, last dimension self.m
An array of points to query.

k : int, optional
The number of nearest neighbors to return.

eps : nonnegative float, optional
Return approximate nearest neighbors; the kth returned value is guaranteed to be no further than (1+eps) times the distance to the real kth nearest neighbor.

p : float, 1<=p<=infinity, optional
Which Minkowski p-norm to use. 1 is the sum-of-absolute-values "Manhattan" distance 2 is the usual Euclidean distance infinity is the maximum-coordinate-difference distance

distance_upper_bound : nonnegative float, optional
Return only neighbors within this distance. This is used to prune tree searches, so if you are doing a series of nearest-neighbor queries, it may help to supply the distance to the nearest neighbor of the most recent point.

Returns

d : float or array of floats
The distances to the nearest neighbors. If x has shape tuple+(self.m,), then d has shape tuple if k is one, or tuple+(k,) if k is larger than one. Missing neighbors (e.g. when k > n or distance_upper_bound is given) are indicated with infinite distances. If k is None, then d is an object array of shape tuple, containing lists of distances. In either case the hits are sorted by distance (nearest first).

i : integer or array of integers
The locations of the neighbors in self.data. i is the same shape as d.

Examples

```
>>> from scipy import spatial
>>> x, y = np.mgrid[0:5, 2:8]
>>> tree = spatial.KDTree(list(zip(x.ravel(), y.ravel())))
>>> tree.data
array([[0, 2],
       [0, 3],
       [0, 4],
       [0, 5],
       [0, 6],
       [0, 7],
       [1, 2],
       [1, 3],
       [1, 4],
       [1, 5],
       [1, 6],
       [1, 7],
       [2, 2],
       [2, 3],
       [2, 4],
       [2, 5],
       [2, 6],
       [2, 7],
       [3, 2],
       [3, 3],
       [3, 4],
```

```
       [3, 5],
       [3, 6],
       [3, 7],
       [4, 2],
       [4, 3],
       [4, 4],
       [4, 5],
       [4, 6],
       [4, 7]])
>>> pts = np.array([[0, 0], [2.1, 2.9]])
>>> tree.query(pts)
(array([ 2.        ,  0.14142136]), array([ 0, 13]))
>>> tree.query(pts[0])
(2.0, 0)
```

KDTree.**query_ball_point**(*x*, *r*, *p=2.0*, *eps=0*)

Find all points within distance r of point(s) x.

Parameters

x : array_like, shape tuple + (self.m,)
The point or points to search for neighbors of.

r : positive float
The radius of points to return.

p : float, optional
Which Minkowski p-norm to use. Should be in the range [1, inf].

eps : nonnegative float, optional
Approximate search. Branches of the tree are not explored if their nearest points are further than `r / (1 + eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1 + eps)`.

Returns

results : list or array of lists
If *x* is a single point, returns a list of the indices of the neighbors of *x*. If *x* is an array of points, returns an object array of shape tuple containing lists of neighbors.

Notes

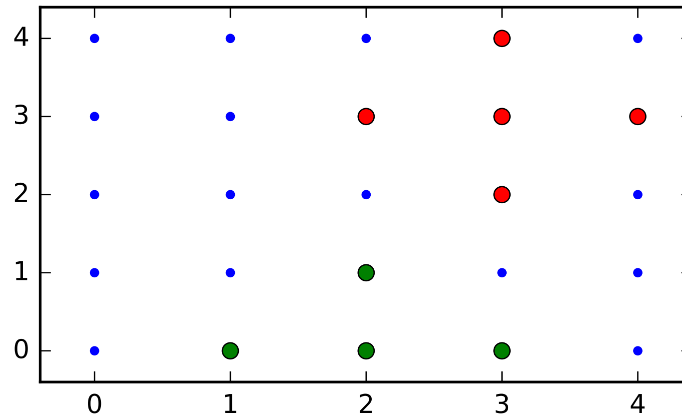
If you have many points whose neighbors you want to find, you may save substantial amounts of time by putting them in a KDTree and using query_ball_tree.

Examples

```
>>> from scipy import spatial
>>> x, y = np.mgrid[0:5, 0:5]
>>> points = zip(x.ravel(), y.ravel())
>>> tree = spatial.KDTree(points)
>>> tree.query_ball_point([2, 0], 1)
[5, 10, 11, 15]
```

Query multiple points and plot the results:

```
>>> import matplotlib.pyplot as plt
>>> points = np.asarray(points)
>>> plt.plot(points[:,0], points[:,1], '.')
>>> for results in tree.query_ball_point(([2, 0], [3, 3]), 1):
...     nearby_points = points[results]
...     plt.plot(nearby_points[:,0], nearby_points[:,1], 'o')
>>> plt.margins(0.1, 0.1)
>>> plt.show()
```



KDTree.**query_ball_tree**(*other*, *r*, *p=2.0*, *eps=0*)
Find all pairs of points whose distance is at most r

Parameters **other** : KDTree instance
The tree containing points to search against.

r : float
The maximum distance, has to be positive.

p : float, optional
Which Minkowski norm to use. *p* has to meet the condition `1 <= p <= infinity`.

eps : float, optional
Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

Returns **results** : list of lists
For each element `self.data[i]` of this tree, `results[i]` is a list of the indices of its neighbors in `other.data`.

KDTree.**query_pairs**(*r*, *p=2.0*, *eps=0*)
Find all pairs of points within a distance.

Parameters **r** : positive float
The maximum distance.

p : float, optional
Which Minkowski norm to use. *p* has to meet the condition `1 <= p <= infinity`.

eps : float, optional
Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

Returns **results** : set
Set of pairs `(i,j)`, with `i < j`, for which the corresponding positions are close.

KDTree.**sparse_distance_matrix**(*other*, *max_distance*, *p=2.0*)

Compute a sparse distance matrix

Computes a distance matrix between two KDTrees, leaving as zero any distance greater than max_distance.

Parameters

other : KDTree

max_distance : positive float

p : float, optional

Returns

result : dok_matrix

Sparse matrix representing the results in "dictionary of keys" format.

class scipy.spatial.**cKDTree**

kd-tree for quick nearest-neighbor lookup

This class provides an index into a set of k-dimensional points which can be used to rapidly look up the nearest neighbors of any point.

The algorithm used is described in Maneewongvatana and Mount 1999. The general idea is that the kd-tree is a binary trie, each of whose nodes represents an axis-aligned hyperrectangle. Each node specifies an axis and splits the set of points based on whether their coordinate along that axis is greater than or less than a particular value.

During construction, the axis and splitting point are chosen by the "sliding midpoint" rule, which ensures that the cells do not all become long and thin.

The tree can be queried for the r closest neighbors of any given point (optionally returning only those within some maximum distance of the point). It can also be queried, with a substantial gain in efficiency, for the r approximate closest neighbors.

For large dimensions (20 is already large) do not expect this to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.

Parameters

data : array_like, shape (n,m)

The n data points of dimension m to be indexed. This array is not copied unless this is necessary to produce a contiguous array of doubles, and so modifying this data will result in bogus results. The data are also copied if the kd-tree is built with copy_data=True.

leafsize : positive int, optional

The number of points at which the algorithm switches over to brute-force. Default: 16.

compact_nodes : bool, optional

If True, the kd-tree is built to shrink the hyperrectangles to the actual data range. This usually gives a more compact tree that is robust against degenerated input data and gives faster queries at the expense of longer build time. Default: True.

copy_data : bool, optional

If True the data is always copied to protect the kd-tree against data corruption. Default: False.

balanced_tree : bool, optional

If True, the median is used to split the hyperrectangles instead of the midpoint. This usually gives a more compact tree and faster queries at the expense of longer build time. Default: True.

boxsize : array_like or scalar, optional

Apply a m-d toroidal topology to the KDTree.. The topology is generated by $x_i + n_i L_i$ where n_i are integers and L_i is the boxsize along i-th dimension. The input data shall be wrapped into $[0, L_i)$. A ValueError is raised if any of the data is outside of this bound.

See also:

KDTree Implementation of *cKDTree* in pure Python

Attributes

data	(ndarray, shape (n,m)) The n data points of dimension m to be indexed. This array is not copied unless this is necessary to produce a contiguous array of doubles. The data are also copied if the kd-tree is built with *copy_data=True*.
leaf-size	(positive int) The number of points at which the algorithm switches over to brute-force.
m	(int) The dimension of a single data-point.
n	(int) The number of data points.
maxes	(ndarray, shape (m,)) The maximum value in each dimension of the n data points.
mins	(ndarray, shape (m,)) The minimum value in each dimension of the n data points.
tree	(object, class cKDTreeNode) This class exposes a Python view of the root node in the cKDTree object.
size	(int) The number of nodes in the tree.

Methods

count_neighbors(self, other, r[, p, ...])	Count how many nearby pairs can be formed.
query(self, x[, k, eps, p, ...])	Query the kd-tree for nearest neighbors
query_ball_point(self, x, r[, p, eps])	Find all points within distance r of point(s) x.
query_ball_tree(self, other, r[, p, eps])	Find all pairs of points whose distance is at most r
query_pairs(self, r[, p, eps])	Find all pairs of points whose distance is at most r.
sparse_distance_matrix(self, other, max_distance)	Compute a sparse distance matrix

cKDTree.**count_neighbors**(*self*, *other*, *r*, *p=2.*, *weights=None*, *cumulative=True*)

Count how many nearby pairs can be formed. (pair-counting)

Count the number of pairs (x1,x2) can be formed, with x1 drawn from self and x2 drawn from `other`, and where `distance(x1, x2, p) <= r`.

Data points on self and other are optionally weighted by the `weights` argument. (See below)

The algorithm we implement here is based on *[R356]*. See notes for further discussion.

Parameters

other : cKDTree instance
: The other tree to draw points from, can be the same tree as self.

r : float or one-dimensional array of floats
: The radius to produce a count for. Multiple radii are searched with a single tree traversal. If the count is non-cumulative(`cumulative=False`), `r` defines the edges of the bins, and must be non-decreasing.

p : float, optional
: 1<=p<=infinity. Which Minkowski p-norm to use. Default 2.0.

weights : tuple, array_like, or None, optional
: If None, the pair-counting is unweighted. If given as a tuple, weights[0] is the weights of points in `self`, and weights[1] is the weights of points in `other`; either can be None to indicate the points are unweighted. If given as an array_like, weights is the weights of points in `self` and `other`. For this to make sense, `self` and `other` must be the same tree. If `self` and `other` are two different trees, a `ValueError` is raised. Default: None

cumulative : bool, optional
Whether the returned counts are cumulative. When cumulative is set to `False` the algorithm is optimized to work with a large number of bins (>10) specified by `r`. When `cumulative` is set to True, the algorithm is optimized to work with a small number of `r`. Default: True

Returns **result** : scalar or 1-D array
The number of pairs. For unweighted counts, the result is integer. For weighted counts, the result is float. If cumulative is False, `result[i]` contains the counts with `(-inf if i == 0 else r[i-1]) < R <= r[i]`

Notes

Pair-counting is the basic operation used to calculate the two point correlation functions from a data set composed of position of objects.

Two point correlation function measures the clustering of objects and is widely used in cosmology to quantify the large scale structure in our Universe, but it may be useful for data analysis in other fields where self-similar assembly of objects also occur.

The Landy-Szalay estimator for the two point correlation function of `D` measures the clustering signal in `D`. *[R357]*

For example, given the position of two sets of objects,

- objects `D` (data) contains the clustering signal, and
- objects `R` (random) that contains no signal,

$$\xi(r) = \frac{<D, D> -2f <D, R> +f^2 <R, R>}{f^2 <R, R>},$$

where the brackets represents counting pairs between two data sets in a finite bin around `r` (distance), corresponding to setting *cumulative=False*, and `f = float(len(D)) / float(len(R))` is the ratio between number of objects from data and random.

The algorithm implemented here is loosely based on the dual-tree algorithm described in *[R356]*. We switch between two different pair-cumulation scheme depending on the setting of `cumulative`. The computing time of the method we use when for `cumulative == False` does not scale with the total number of bins. The algorithm for `cumulative == True` scales linearly with the number of bins, though it is slightly faster when only 1 or 2 bins are used. *[R360]*.

As an extension to the naive pair-counting, weighted pair-counting counts the product of weights instead of number of pairs. Weighted pair-counting is used to estimate marked correlation functions (*[R358]*, section 2.2), or to properly calculate the average of data per distance bin (e.g. *[R359]*, section 2.1 on redshift).

`cKDTree.`**`query`**(*self*, *x*, *k=1*, *eps=0*, *p=2*, *distance_upper_bound=np.inf*, *n_jobs=1*)
Query the kd-tree for nearest neighbors

Parameters **x** : array_like, last dimension self.m
An array of points to query.

k : list of integer or integer
The list of k-th nearest neighbors to return. If k is an integer it is treated as a list of [1, ... k] (range(1, k+1)). Note that the counting starts from 1.

eps : non-negative float
Return approximate nearest neighbors; the k-th returned value is guaranteed to be no further than (1+eps) times the distance to the real k-th nearest neighbor.

p : float, 1<=p<=infinity

Which Minkowski p-norm to use. 1 is the sum-of-absolute-values "Manhattan" distance 2 is the usual Euclidean distance infinity is the maximum-coordinate-difference distance

distance_upper_bound : nonnegative float
Return only neighbors within this distance. This is used to prune tree searches, so if you are doing a series of nearest-neighbor queries, it may help to supply the distance to the nearest neighbor of the most recent point.

n_jobs : int, optional
Number of jobs to schedule for parallel processing. If -1 is given all processors are used. Default: 1.

Returns

d : array of floats
The distances to the nearest neighbors. If `x` has shape `tuple+(self.m,)`, then `d` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with infinite distances.

i : ndarray of ints
The locations of the neighbors in `self.data`. If `x` has shape `tuple+(self.m,)`, then `i` has shape `tuple+(k,)`. When k == 1, the last dimension of the output is squeezed. Missing neighbors are indicated with `self.n`.

Notes

If the KD-Tree is periodic, the position `x` is wrapped into the box.

When the input k is a list, a query for arange(max(k)) is performed, but only columns that store the requested values of k are preserved. This is implemented in a manner that reduces memory usage.

Examples

```
>>> tree = cKDTree(data)
```

To query the nearest neighbours and return squeezed result, use

```
>>> dd, ii = tree.query(x, k=1)
```

To query the nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query(x, k=[1])
```

To query the second nearest neighbours and return unsqueezed result, use

```
>>> dd, ii = tree.query(x, k=[2])
```

To query the first and second nearest neighbours, use

```
>>> dd, ii = tree.query(x, k=2)
```

or, be more specific

```
>>> dd, ii = tree.query(x, k=[1, 2])
```

`cKDTree.`**`query_ball_point`**(*self*, *x*, *r*, *p=2.*, *eps=0*)

Find all points within distance r of point(s) x.

Parameters

x : array_like, shape tuple + (self.m,)
The point or points to search for neighbors of.

r : positive float
: The radius of points to return.

p : float, optional
: Which Minkowski p-norm to use. Should be in the range [1, inf].

eps : nonnegative float, optional
: Approximate search. Branches of the tree are not explored if their nearest points are further than `r / (1 + eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1 + eps)`.

n_jobs : int, optional
: Number of jobs to schedule for parallel processing. If -1 is given all processors are used. Default: 1.

Returns **results** : list or array of lists
: If *x* is a single point, returns a list of the indices of the neighbors of *x*. If *x* is an array of points, returns an object array of shape tuple containing lists of neighbors.

Notes

If you have many points whose neighbors you want to find, you may save substantial amounts of time by putting them in a cKDTree and using query_ball_tree.

Examples

```
>>> from scipy import spatial
>>> x, y = np.mgrid[0:4, 0:4]
>>> points = zip(x.ravel(), y.ravel())
>>> tree = spatial.cKDTree(points)
>>> tree.query_ball_point([2, 0], 1)
[4, 8, 9, 12]
```

`cKDTree.`**`query_ball_tree`**(*self*, *other*, *r*, *p=2.*, *eps=0*)

Find all pairs of points whose distance is at most r

Parameters **other** : cKDTree instance
: The tree containing points to search against.

r : float
: The maximum distance, has to be positive.

p : float, optional
: Which Minkowski norm to use. *p* has to meet the condition `1 <= p <= infinity`.

eps : float, optional
: Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

Returns **results** : list of lists
: For each element `self.data[i]` of this tree, `results[i]` is a list of the indices of its neighbors in `other.data`.

`cKDTree.`**`query_pairs`**(*self*, *r*, *p=2.*, *eps=0*)

Find all pairs of points whose distance is at most r.

Parameters **r** : positive float
: The maximum distance.

p : float, optional
: Which Minkowski norm to use. `p` has to meet the condition `1 <= p <= infinity`.

eps : float, optional
Approximate search. Branches of the tree are not explored if their nearest points are further than `r/(1+eps)`, and branches are added in bulk if their furthest points are nearer than `r * (1+eps)`. *eps* has to be non-negative.

output_type : string, optional
Choose the output container, 'set' or 'ndarray'. Default: 'set'

Returns **results** : set or ndarray
Set of pairs `(i,j)`, with `i < j`, for which the corresponding positions are close. If output_type is 'ndarray', an ndarry is returned instead of a set.

`cKDTree.`**`sparse_distance_matrix`**(*self*, *other*, *max_distance*, *p=2.*)
Compute a sparse distance matrix

Computes a distance matrix between two cKDTrees, leaving as zero any distance greater than max_distance.

Parameters **other** : cKDTree

max_distance : positive float

p : float, 1<=p<=infinity
Which Minkowski p-norm to use.

output_type : string, optional
Which container to use for output data. Options: 'dok_matrix', 'coo_matrix', 'dict', or 'ndarray'. Default: 'dok_matrix'.

Returns **result** : dok_matrix, coo_matrix, dict or ndarray
Sparse matrix representing the results in "dictionary of keys" format. If a dict is returned the keys are (i,j) tuples of indices. If output_type is 'ndarray' a record array with fields 'i', 'j', and 'k' is returned,

Distance computations (`scipy.spatial.distance`)

Function Reference

Distance matrix computation from a collection of raw observation vectors stored in a rectangular array.

pdist(X[, metric, p, w, V, VI])	Pairwise distances between observations in n-dimensional space.
cdist(XA, XB[, metric, p, V, VI, w])	Computes distance between each pair of the two collections of inputs.
squareform(X[, force, checks])	Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.
directed_hausdorff(u, v[, seed])	Computes the directed Hausdorff distance between two N-D arrays.

`scipy.spatial.distance.`**`pdist`**(*X*, *metric='euclidean'*, *p=None*, *w=None*, *V=None*, *VI=None*)
Pairwise distances between observations in n-dimensional space.

See Notes for common calling conventions.

Parameters **X** : ndarray
An m by n array of m original observations in an n-dimensional space.

metric : str or function, optional
The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching',

'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.

p : double, optional
The p-norm to apply Only for Minkowski, weighted and unweighted. Default: 2.

w : ndarray, optional
The weight vector. Only for weighted Minkowski. Mandatory

V : ndarray, optional
The variance vector Only for standardized Euclidean. Default: var(X, axis=0, ddof=1)

VI : ndarray, optional
The inverse of the covariance matrix Only for Mahalanobis. Default: inv(cov(X.T)).T

Returns **Y** : ndarray
Returns a condensed distance matrix Y. For each i and j (where $i < j < m$),where m is the number of original observations. The metric `dist(u=X[i], v=X[j])` is computed and stored in entry `ij`.

See also:

`squareform`converts between condensed distance matrices and square distance matrices.

Notes

See `squareform` for information on how to calculate the index of this entry or to convert the condensed distance matrix to a redundant square matrix.

The following are common calling conventions.

1. `Y = pdist(X, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = pdist(X, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = pdist(X, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = pdist(X, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

$$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}$$

   V is the variance vector; V[i] is the variance computed over all the i'th components of the points. If not passed, it is automatically computed.

5. `Y = pdist(X, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = pdist(X, 'cosine')`

   Computes the cosine distance between vectors u and v,

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

   where $||*||_2$ is the 2-norm of its argument `*`, and $u \cdot v$ is the dot product of `u` and `v`.

7. `Y = pdist(X, 'correlation')`

Computes the correlation distance between vectors u and v. This is

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

where $\bar{v}$ is the mean of the elements of vector v, and $x \cdot y$ is the dot product of x and y.

8. `Y = pdist(X, 'hamming')`

Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = pdist(X, 'jaccard')`

Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree.

10. `Y = pdist(X, 'chebyshev')`

Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

$$d(u, v) = \max_i |u_i - v_i|$$

11. `Y = pdist(X, 'canberra')`

Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

$$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}$$

12. `Y = pdist(X, 'braycurtis')`

Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

$$d(u, v) = \frac{\sum_i |u_i - v_i|}{\sum_i |u_i + v_i|}$$

13. `Y = pdist(X, 'mahalanobis', VI=None)`

Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $\sqrt{(u - v)(1/V)(u - v)^T}$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

14. `Y = pdist(X, 'yule')`

Computes the Yule distance between each pair of boolean vectors. (see yule function documentation)

15. `Y = pdist(X, 'matching')`

Synonym for 'hamming'.

16. `Y = pdist(X, 'dice')`

Computes the Dice distance between each pair of boolean vectors. (see dice function documentation)

17. `Y = pdist(X, 'kulsinski')`

Computes the Kulsinski distance between each pair of boolean vectors. (see kulsinski function documentation)

18. `Y = pdist(X, 'rogerstanimoto')`

    Computes the Rogers-Tanimoto distance between each pair of boolean vectors. (see rogerstanimoto function documentation)

19. `Y = pdist(X, 'russellrao')`

    Computes the Russell-Rao distance between each pair of boolean vectors. (see russellrao function documentation)

20. `Y = pdist(X, 'sokalmichener')`

    Computes the Sokal-Michener distance between each pair of boolean vectors. (see sokalmichener function documentation)

21. `Y = pdist(X, 'sokalsneath')`

    Computes the Sokal-Sneath distance between each pair of boolean vectors. (see sokalsneath function documentation)

22. `Y = pdist(X, 'wminkowski')`

    Computes the weighted Minkowski distance between each pair of vectors. (see wminkowski function documentation)

23. `Y = pdist(X, f)`

    Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

```
dm = pdist(X, lambda u, v: np.sqrt(((u-v)**2).sum()))
```

Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

```
dm = pdist(X, sokalsneath)
```

would calculate the pair-wise distances between the vectors in X using the Python function sokalsneath. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax.:

```
dm = pdist(X, 'sokalsneath')
```

`scipy.spatial.distance.`**`cdist`**(*XA*, *XB*, *metric='euclidean'*, *p=None*, *V=None*, *VI=None*, *w=None*)

Computes distance between each pair of the two collections of inputs.

See Notes for common calling conventions.

Parameters

XA : ndarray

An m_A by n array of m_A original observations in an n-dimensional space. Inputs are converted to float type.

XB : ndarray

An m_B by n array of m_B original observations in an n-dimensional space. Inputs are converted to float type.

metric : str or callable, optional

The distance metric to use. If a string, the distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

p : double, optional
The p-norm to apply Only for Minkowski, weighted and unweighted. Default: 2.

w : ndarray, optional
The weight vector. Only for weighted Minkowski. Mandatory

V : ndarray, optional
The variance vector Only for standardized Euclidean. Default: var(vstack([XA, XB]), axis=0, ddof=1)

VI : ndarray, optional
The inverse of the covariance matrix Only for Mahalanobis. Default: inv(cov(vstack([XA, XB]).T)).T

Returns **Y** : ndarray
A m_A by m_B distance matrix is returned. For each i and j, the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the ij th entry.

Raises **ValueError**
An exception is thrown if *XA* and *XB* do not have the same number of columns.

Notes

The following are common calling conventions:

1. `Y = cdist(XA, XB, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = cdist(XA, XB, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = cdist(XA, XB, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = cdist(XA, XB, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

$$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}.$$

   V is the variance vector; V[i] is the variance computed over all the i'th components of the points. If not passed, it is automatically computed.

5. `Y = cdist(XA, XB, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = cdist(XA, XB, 'cosine')`

   Computes the cosine distance between vectors u and v,

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

   where $||*||_2$ is the 2-norm of its argument $*$, and $u \cdot v$ is the dot product of u and v.

7. `Y = cdist(XA, XB, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

where $\bar{v}$ is the mean of the elements of vector v, and $x \cdot y$ is the dot product of x and y.

8. `Y = cdist(XA, XB, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = cdist(XA, XB, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree where at least one of them is non-zero.

10. `Y = cdist(XA, XB, 'chebyshev')`

    Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

    $$d(u, v) = \max_i |u_i - v_i|.$$

11. `Y = cdist(XA, XB, 'canberra')`

    Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

    $$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}.$$

12. `Y = cdist(XA, XB, 'braycurtis')`

    Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

    $$d(u, v) = \frac{\sum_i (|u_i - v_i|)}{\sum_i (|u_i + v_i|)}$$

13. `Y = cdist(XA, XB, 'mahalanobis', VI=None)`

    Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $\sqrt{(u - v)(1/V)(u - v)^T}$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

14. `Y = cdist(XA, XB, 'yule')`

    Computes the Yule distance between the boolean vectors. (see *yule* function documentation)

15. `Y = cdist(XA, XB, 'matching')`

    Synonym for 'hamming'.

16. `Y = cdist(XA, XB, 'dice')`

    Computes the Dice distance between the boolean vectors. (see *dice* function documentation)

17. `Y = cdist(XA, XB, 'kulsinski')`

    Computes the Kulsinski distance between the boolean vectors. (see *kulsinski* function documentation)

18. `Y = cdist(XA, XB, 'rogerstanimoto')`

    Computes the Rogers-Tanimoto distance between the boolean vectors. (see *rogerstanimoto* function documentation)

19. `Y = cdist(XA, XB, 'russellrao')`

    Computes the Russell-Rao distance between the boolean vectors. (see *russellrao* function documentation)

20. `Y = cdist(XA, XB, 'sokalmichener')`

    Computes the Sokal-Michener distance between the boolean vectors. (see *sokalmichener* function documentation)

21. `Y = cdist(XA, XB, 'sokalsneath')`

    Computes the Sokal-Sneath distance between the vectors. (see *sokalsneath* function documentation)

22. `Y = cdist(XA, XB, 'wminkowski')`

    Computes the weighted Minkowski distance between the vectors. (see *wminkowski* function documentation)

23. `Y = cdist(XA, XB, f)`

    Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

    ```
    dm = cdist(XA, XB, lambda u, v: np.sqrt(((u-v)**2).sum()))
    ```

    Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

    ```
    dm = cdist(XA, XB, sokalsneath)
    ```

    would calculate the pair-wise distances between the vectors in X using the Python function *sokalsneath*. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax:

    ```
    dm = cdist(XA, XB, 'sokalsneath')
    ```

Examples

Find the Euclidean distances between four 2-D coordinates:

```
>>> from scipy.spatial import distance
>>> coords = [(35.0456, -85.2672),
...           (35.1174, -89.9711),
...           (35.9728, -83.9422),
...           (36.1667, -86.7833)]
>>> distance.cdist(coords, coords, 'euclidean')
array([[ 0.    ,  4.7044,  1.6172,  1.8856],
       [ 4.7044,  0.    ,  6.0893,  3.3561],
       [ 1.6172,  6.0893,  0.    ,  2.8477],
       [ 1.8856,  3.3561,  2.8477,  0.    ]])
```

Find the Manhattan distance from a 3-D point to the corners of the unit cube:

```
>>> a = np.array([[0, 0, 0],
...               [0, 0, 1],
...               [0, 1, 0],
...               [0, 1, 1],
...               [1, 0, 0],
```

```
...                [1, 0, 1],
...                [1, 1, 0],
...                [1, 1, 1]])
>>> b = np.array([[ 0.1,  0.2,  0.4]])
>>> distance.cdist(a, b, 'cityblock')
array([[ 0.7],
       [ 0.9],
       [ 1.3],
       [ 1.5],
       [ 1.5],
       [ 1.7],
       [ 2.1],
       [ 2.3]])
```

scipy.spatial.distance.**squareform**(*X*, *force='no'*, *checks=True*)

Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.

Parameters **X** : ndarray

Either a condensed or redundant distance matrix.

force : str, optional

As with MATLAB(TM), if force is equal to `'tovector'` or `'tomatrix'`, the input will be treated as a distance matrix or distance vector respectively.

checks : bool, optional

If set to False, no checks will be made for matrix symmetry nor zero diagonals. This is useful if it is known that `X - X.T1` is small and `diag(X)` is close to zero. These values are ignored any way so they do not disrupt the squareform transformation.

Returns **Y** : ndarray

If a condensed distance matrix is passed, a redundant one is returned, or if a redundant one is passed, a condensed distance matrix is returned.

Notes

1. v = squareform(X)

Given a square d-by-d symmetric distance matrix X, `v = squareform(X)` returns a `d * (d-1) / 2` (or $\binom{n}{2}$) sized vector v.

$v[\binom{n}{2} - \binom{n-i}{2} + (j - i - 1)]$ is the distance between points i and j. If X is non-square or asymmetric, an error is returned.

2. X = squareform(v)

Given a `d*(d-1)/2` sized v for some integer `d >= 2` encoding distances as described, `X = squareform(v)` returns a d by d distance matrix X. The `X[i, j]` and `X[j, i]` values are set to $v[\binom{n}{2} - \binom{n-i}{2} + (j - i - 1)]$ and all diagonal elements are zero.

In Scipy 0.19.0, `squareform` stopped casting all input types to float64, and started returning arrays of the same dtype as the input.

scipy.spatial.distance.**directed_hausdorff**(*u*, *v*, *seed=0*)

Computes the directed Hausdorff distance between two N-D arrays.

Distances between pairs are calculated using a Euclidean metric.

Parameters **u** : (M,N) ndarray

Input array.

v : (O,N) ndarray
: Input array.

seed : int or None
: Local `np.random.RandomState` seed. Default is 0, a random shuffling of u and v that guarantees reproducibility.

Returns

d : double
: The directed Hausdorff distance between arrays *u* and *v*,

index_1 : int
: index of point contributing to Hausdorff pair in *u*

index_2 : int
: index of point contributing to Hausdorff pair in *v*

See also:

`scipy.spatial.procrustes`
: Another similarity test for two data sets

Notes

Uses the early break technique and the random sampling approach described by *[R40]*. Although worst-case performance is `O(m * o)` (as with the brute force algorithm), this is unlikely in practice as the input data would have to require the algorithm to explore every single point interaction, and after the algorithm shuffles the input points at that. The best case performance is O(m), which is satisfied by selecting an inner loop distance that is less than cmax and leads to an early break as often as possible. The authors have formally shown that the average runtime is closer to O(m).

New in version 0.19.0.

References

[R40]

Examples

Find the directed Hausdorff distance between two 2-D arrays of coordinates:

```
>>> from scipy.spatial.distance import directed_hausdorff
>>> u = np.array([(1.0, 0.0),
...               (0.0, 1.0),
...               (-1.0, 0.0),
...               (0.0, -1.0)])
>>> v = np.array([(2.0, 0.0),
...               (0.0, 2.0),
...               (-2.0, 0.0),
...               (0.0, -4.0)])
```

```
>>> directed_hausdorff(u, v)[0]
2.23606797749979
>>> directed_hausdorff(v, u)[0]
3.0
```

Find the general (symmetric) Hausdorff distance between two 2-D arrays of coordinates:

```
>>> max(directed_hausdorff(u, v)[0], directed_hausdorff(v, u)[0])
3.0
```

Find the indices of the points that generate the Hausdorff distance (the Hausdorff pair):

```
>>> directed_hausdorff(v, u)[1:]
(3, 3)
```

Predicates for checking the validity of distance matrices, both condensed and redundant. Also contained in this module are functions for computing the number of observations in a distance matrix.

is_valid_dm(D[, tol, throw, name, warning])	Returns True if input array is a valid distance matrix.
is_valid_y(y[, warning, throw, name])	Returns True if the input array is a valid condensed distance matrix.
num_obs_dm(d)	Returns the number of original observations that correspond to a square, redundant distance matrix.
num_obs_y(Y)	Returns the number of original observations that correspond to a condensed distance matrix.

scipy.spatial.distance.**is_valid_dm**(*D*, *tol=0.0*, *throw=False*, *name='D'*, *warning=False*)

Returns True if input array is a valid distance matrix.

Distance matrices must be 2-dimensional numpy arrays. They must have a zero-diagonal, and they must be symmetric.

Parameters

D : ndarray

The candidate object to test for validity.

tol : float, optional

The distance matrix should be symmetric. *tol* is the maximum difference between entries `ij` and `ji` for the distance metric to be considered symmetric.

throw : bool, optional

An exception is thrown if the distance matrix passed is not valid.

name : str, optional

The name of the variable to checked. This is useful if throw is set to True so the offending variable can be identified in the exception message when an exception is thrown.

warning : bool, optional

Instead of throwing an exception, a warning message is raised.

Returns

valid : bool

True if the variable *D* passed is a valid distance matrix.

Notes

Small numerical differences in *D* and *D.T* and non-zeroness of the diagonal are ignored if they are within the tolerance specified by *tol*.

scipy.spatial.distance.**is_valid_y**(*y*, *warning=False*, *throw=False*, *name=None*)

Returns True if the input array is a valid condensed distance matrix.

Condensed distance matrices must be 1-dimensional numpy arrays. Their length must be a binomial coefficient $\binom{n}{2}$ for some positive integer n.

Parameters

y : ndarray

The condensed distance matrix.

warning : bool, optional

Invokes a warning if the variable passed is not a valid condensed distance matrix. The warning message explains why the distance matrix is not valid. `name` is used when referencing the offending variable.

throw : bool, optional

Throws an exception if the variable passed is not a valid condensed distance matrix.

name : bool, optional

Used when referencing the offending variable in the warning or exception message.

`scipy.spatial.distance.`**`num_obs_dm`**`(d)`

Returns the number of original observations that correspond to a square, redundant distance matrix.

Parameters **d** : ndarray

The target distance matrix.

Returns **num_obs_dm** : int

The number of observations in the redundant distance matrix.

`scipy.spatial.distance.`**`num_obs_y`**`(Y)`

Returns the number of original observations that correspond to a condensed distance matrix.

Parameters **Y** : ndarray

Condensed distance matrix.

Returns **n** : int

The number of observations in the condensed distance matrix *Y*.

Distance functions between two numeric vectors `u` and `v`. Computing distances over a large collection of vectors is inefficient for these functions. Use `pdist` for this purpose.

braycurtis(u, v)	Computes the Bray-Curtis distance between two 1-D arrays.
canberra(u, v)	Computes the Canberra distance between two 1-D arrays.
chebyshev(u, v)	Computes the Chebyshev distance.
cityblock(u, v)	Computes the City Block (Manhattan) distance.
correlation(u, v)	Computes the correlation distance between two 1-D arrays.
cosine(u, v)	Computes the Cosine distance between 1-D arrays.
euclidean(u, v)	Computes the Euclidean distance between two 1-D arrays.
mahalanobis(u, v, VI)	Computes the Mahalanobis distance between two 1-D arrays.
minkowski(u, v, p)	Computes the Minkowski distance between two 1-D arrays.
seuclidean(u, v, V)	Returns the standardized Euclidean distance between two 1-D arrays.
sqeuclidean(u, v)	Computes the squared Euclidean distance between two 1-D arrays.
wminkowski(u, v, p, w)	Computes the weighted Minkowski distance between two 1-D arrays.

`scipy.spatial.distance.`**`braycurtis`**`(u, v)`

Computes the Bray-Curtis distance between two 1-D arrays.

Bray-Curtis distance is defined as

$$\sum |u_i - v_i| / \sum |u_i + v_i|$$

The Bray-Curtis distance is in the range [0, 1] if all coordinates are positive, and is undefined if the inputs are of length zero.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **braycurtis** : double

The Bray-Curtis distance between 1-D arrays *u* and *v*.

`scipy.spatial.distance.canberra(u, v)`

Computes the Canberra distance between two 1-D arrays.

The Canberra distance is defined as

$$d(u,v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}.$$

Parameters
- **u** : (N,) array_like
  Input array.
- **v** : (N,) array_like
  Input array.

Returns
- **canberra** : double
  The Canberra distance between vectors *u* and *v*.

Notes

When *u[i]* and *v[i]* are 0 for given i, then the fraction 0/0 = 0 is used in the calculation.

`scipy.spatial.distance.chebyshev(u, v)`

Computes the Chebyshev distance.

Computes the Chebyshev distance between two 1-D arrays *u* and *v*, which is defined as

$$\max_i |u_i - v_i|.$$

Parameters
- **u** : (N,) array_like
  Input vector.
- **v** : (N,) array_like
  Input vector.

Returns
- **chebyshev** : double
  The Chebyshev distance between vectors *u* and *v*.

`scipy.spatial.distance.cityblock(u, v)`

Computes the City Block (Manhattan) distance.

Computes the Manhattan distance between two 1-D arrays *u* and *v*, which is defined as

$$\sum_i |u_i - v_i|.$$

Parameters
- **u** : (N,) array_like
  Input array.
- **v** : (N,) array_like
  Input array.

Returns
- **cityblock** : double
  The City Block (Manhattan) distance between vectors *u* and *v*.

`scipy.spatial.distance.correlation(u, v)`

Computes the correlation distance between two 1-D arrays.

The correlation distance between *u* and *v*, is defined as

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

where $\bar{u}$ is the mean of the elements of *u* and $x \cdot y$ is the dot product of x and y.

Parameters
- **u** : (N,) array_like
  Input array.

v : (N,) array_like
Input array.

Returns **correlation** : double
The correlation distance between 1-D array *u* and *v*.

`scipy.spatial.distance.`**`cosine`**(*u*, *v*)
Computes the Cosine distance between 1-D arrays.

The Cosine distance between *u* and *v*, is defined as

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2}.$$

where $u \cdot v$ is the dot product of u and v.

Parameters **u** : (N,) array_like
Input array.
v : (N,) array_like
Input array.

Returns **cosine** : double
The Cosine distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`euclidean`**(*u*, *v*)
Computes the Euclidean distance between two 1-D arrays.

The Euclidean distance between 1-D arrays *u* and *v*, is defined as

$$||u - v||_2$$

Parameters **u** : (N,) array_like
Input array.
v : (N,) array_like
Input array.

Returns **euclidean** : double
The Euclidean distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`mahalanobis`**(*u*, *v*, *VI*)
Computes the Mahalanobis distance between two 1-D arrays.

The Mahalanobis distance between 1-D arrays *u* and *v*, is defined as

$$\sqrt{(u - v) V^{-1} (u - v)^T}$$

where `V` is the covariance matrix. Note that the argument *VI* is the inverse of `V`.

Parameters **u** : (N,) array_like
Input array.
v : (N,) array_like
Input array.
VI : ndarray
The inverse of the covariance matrix.

Returns **mahalanobis** : double
The Mahalanobis distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`minkowski`**(*u*, *v*, *p*)
Computes the Minkowski distance between two 1-D arrays.

The Minkowski distance between 1-D arrays *u* and *v*, is defined as

$$||u - v||_p = (\sum |u_i - v_i|^p)^{1/p}.$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

p : int

The order of the norm of the difference $||u - v||_p$.

Returns **d** : double

The Minkowski distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`seuclidean`**(*u*, *v*, *V*)

Returns the standardized Euclidean distance between two 1-D arrays.

The standardized Euclidean distance between *u* and *v*.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

V : (N,) array_like

V is an 1-D array of component variances. It is usually computed among a larger collection vectors.

Returns **seuclidean** : double

The standardized Euclidean distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`sqeuclidean`**(*u*, *v*)

Computes the squared Euclidean distance between two 1-D arrays.

The squared Euclidean distance between *u* and *v* is defined as

$$||u - v||_2^2.$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **sqeuclidean** : double

The squared Euclidean distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`wminkowski`**(*u*, *v*, *p*, *w*)

Computes the weighted Minkowski distance between two 1-D arrays.

The weighted Minkowski distance between *u* and *v*, defined as

$$\left(\sum (|w_i(u_i - v_i)|^p)\right)^{1/p}.$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

p : int

The order of the norm of the difference $||u - v||_p$.

w : (N,) array_like

The weight vector.

Returns **wminkowski** : double

The weighted Minkowski distance between vectors *u* and *v*.

Distance functions between two boolean vectors (representing sets) `u` and `v`. As in the case of numerical vectors, `pdist` is more efficient for computing the distances between all pairs.

dice(u, v)	Computes the Dice dissimilarity between two boolean 1-D arrays.
hamming(u, v)	Computes the Hamming distance between two 1-D arrays.
jaccard(u, v)	Computes the Jaccard-Needham dissimilarity between two boolean 1-D arrays.
kulsinski(u, v)	Computes the Kulsinski dissimilarity between two boolean 1-D arrays.
matching(u, v)	Computes the Hamming distance between two boolean 1-D arrays.
rogerstanimoto(u, v)	Computes the Rogers-Tanimoto dissimilarity between two boolean 1-D arrays.
russellrao(u, v)	Computes the Russell-Rao dissimilarity between two boolean 1-D arrays.
sokalmichener(u, v)	Computes the Sokal-Michener dissimilarity between two boolean 1-D arrays.
sokalsneath(u, v)	Computes the Sokal-Sneath dissimilarity between two boolean 1-D arrays.
yule(u, v)	Computes the Yule dissimilarity between two boolean 1-D arrays.

scipy.spatial.distance.**dice**(*u*, *v*)

Computes the Dice dissimilarity between two boolean 1-D arrays.

The Dice dissimilarity between *u* and *v*, is

$$\frac{c_{TF} + c_{FT}}{2c_{TT} + c_{FT} + c_{TF}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u : (N,) ndarray, bool
    Input 1-D array.

v : (N,) ndarray, bool
    Input 1-D array.

Returns

dice : double
    The Dice dissimilarity between 1-D arrays *u* and *v*.

scipy.spatial.distance.**hamming**(*u*, *v*)

Computes the Hamming distance between two 1-D arrays.

The Hamming distance between 1-D arrays *u* and *v*, is simply the proportion of disagreeing components in *u* and *v*. If *u* and *v* are boolean vectors, the Hamming distance is

$$\frac{c_{01} + c_{10}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters

u : (N,) array_like
    Input array.

v : (N,) array_like
    Input array.

Returns

hamming : double
    The Hamming distance between vectors *u* and *v*.

scipy.spatial.distance.**jaccard**(*u*, *v*)

Computes the Jaccard-Needham dissimilarity between two boolean 1-D arrays.

The Jaccard-Needham dissimilarity between 1-D boolean arrays *u* and *v*, is defined as

$$\frac{c_{TF} + c_{FT}}{c_{TT} + c_{FT} + c_{TF}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters
- **u** : (N,) array_like, bool
  Input array.
- **v** : (N,) array_like, bool
  Input array.

Returns
- **jaccard** : double
  The Jaccard distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`kulsinski`**`(u, v)`

Computes the Kulsinski dissimilarity between two boolean 1-D arrays.

The Kulsinski dissimilarity between two boolean 1-D arrays *u* and *v*, is defined as

$$\frac{c_{TF} + c_{FT} - c_{TT} + n}{c_{FT} + c_{TF} + n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters
- **u** : (N,) array_like, bool
  Input array.
- **v** : (N,) array_like, bool
  Input array.

Returns
- **kulsinski** : double
  The Kulsinski distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`matching`**`(u, v)`

Computes the Hamming distance between two boolean 1-D arrays.

This is a deprecated synonym for *`hamming`*.

`scipy.spatial.distance.`**`rogerstanimoto`**`(u, v)`

Computes the Rogers-Tanimoto dissimilarity between two boolean 1-D arrays.

The Rogers-Tanimoto dissimilarity between two boolean 1-D arrays *u* and *v*, is defined as

$$\frac{R}{c_{TT} + c_{FF} + R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters
- **u** : (N,) array_like, bool
  Input array.
- **v** : (N,) array_like, bool
  Input array.

Returns
- **rogerstanimoto** : double
  The Rogers-Tanimoto dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`russellrao`**`(u, v)`

Computes the Russell-Rao dissimilarity between two boolean 1-D arrays.

The Russell-Rao dissimilarity between two boolean 1-D arrays, *u* and *v*, is defined as

$$\frac{n - c_{TT}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters
- **u** : (N,) array_like, bool

Input array.

v : (N,) array_like, bool
Input array.

Returns **russellrao** : double
The Russell-Rao dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`sokalmichener`**(*u*, *v*)

Computes the Sokal-Michener dissimilarity between two boolean 1-D arrays.

The Sokal-Michener dissimilarity between boolean 1-D arrays *u* and *v*, is defined as

$$\frac{R}{S+R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$, $R = 2 * (c_{TF} + c_{FT})$ and $S = c_{FF} + c_{TT}$.

Parameters **u** : (N,) array_like, bool
Input array.

v : (N,) array_like, bool
Input array.

Returns **sokalmichener** : double
The Sokal-Michener dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`sokalsneath`**(*u*, *v*)

Computes the Sokal-Sneath dissimilarity between two boolean 1-D arrays.

The Sokal-Sneath dissimilarity between *u* and *v*,

$$\frac{R}{c_{TT}+R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters **u** : (N,) array_like, bool
Input array.

v : (N,) array_like, bool
Input array.

Returns **sokalsneath** : double
The Sokal-Sneath dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`yule`**(*u*, *v*)

Computes the Yule dissimilarity between two boolean 1-D arrays.

The Yule dissimilarity is defined as

$$\frac{R}{c_{TT} * c_{FF} + \frac{R}{2}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2.0 * c_{TF} * c_{FT}$.

Parameters **u** : (N,) array_like, bool
Input array.

v : (N,) array_like, bool
Input array.

Returns **yule** : double
The Yule dissimilarity between vectors *u* and *v*.

`hamming` also operates over discrete numerical vectors.

Functions

braycurtis(u, v)	Computes the Bray-Curtis distance between two 1-D arrays.
callable((object) -> bool)	Return whether the object is callable (i.e., some kind of function).
canberra(u, v)	Computes the Canberra distance between two 1-D arrays.
cdist(XA, XB[, metric, p, V, VI, w])	Computes distance between each pair of the two collections of inputs.
cdist_fn	
chebyshev(u, v)	Computes the Chebyshev distance.
cityblock(u, v)	Computes the City Block (Manhattan) distance.
converter(X)	
correlation(u, v)	Computes the correlation distance between two 1-D arrays.
cosine(u, v)	Computes the Cosine distance between 1-D arrays.
dice(u, v)	Computes the Dice dissimilarity between two boolean 1-D arrays.
directed_hausdorff(u, v[, seed])	Computes the directed Hausdorff distance between two N-D arrays.
euclidean(u, v)	Computes the Euclidean distance between two 1-D arrays.
hamming(u, v)	Computes the Hamming distance between two 1-D arrays.
is_valid_dm(D[, tol, throw, name, warning])	Returns True if input array is a valid distance matrix.
is_valid_y(y[, warning, throw, name])	Returns True if the input array is a valid condensed distance matrix.
jaccard(u, v)	Computes the Jaccard-Needham dissimilarity between two boolean 1-D arrays.
kulsinski(u, v)	Computes the Kulsinski dissimilarity between two boolean 1-D arrays.
mahalanobis(u, v, VI)	Computes the Mahalanobis distance between two 1-D arrays.
matching(u, v)	Computes the Hamming distance between two boolean 1-D arrays.
minkowski(u, v, p)	Computes the Minkowski distance between two 1-D arrays.
norm(a[, ord, axis, keepdims])	Matrix or vector norm.
num_obs_dm(d)	Returns the number of original observations that correspond to a square, redundant distance matrix.
num_obs_y(Y)	Returns the number of original observations that correspond to a condensed distance matrix.
pdist(X[, metric, p, w, V, VI])	Pairwise distances between observations in n-dimensional space.
pdist_fn	
rogerstanimoto(u, v)	Computes the Rogers-Tanimoto dissimilarity between two boolean 1-D arrays.
russellrao(u, v)	Computes the Russell-Rao dissimilarity between two boolean 1-D arrays.
seuclidean(u, v, V)	Returns the standardized Euclidean distance between two 1-D arrays.
sokalmichener(u, v)	Computes the Sokal-Michener dissimilarity between two boolean 1-D arrays.
sokalsneath(u, v)	Computes the Sokal-Sneath dissimilarity between two boolean 1-D arrays.
sqeuclidean(u, v)	Computes the squared Euclidean distance between two 1-D arrays.

Continued on next page

Table 5.214 – continued from previous page

squareform(X[, force, checks])	Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.
wminkowski(u, v, p, w)	Computes the weighted Minkowski distance between two 1-D arrays.
yule(u, v)	Computes the Yule dissimilarity between two boolean 1-D arrays.

Classes

partial	partial(func, *args, **keywords) - new function with partial application
xrange	xrange(start, stop[, step]) -> xrange object

class scipy.spatial.**Rectangle**(*maxes*, *mins*)

Hyperrectangle class.

Represents a Cartesian product of intervals.

Methods

max_distance_point(x[, p])	Return the maximum distance between input and points in the hyperrectangle.
max_distance_rectangle(other[, p])	Compute the maximum distance between points in the two hyperrectangles.
min_distance_point(x[, p])	Return the minimum distance between input and points in the hyperrectangle.
min_distance_rectangle(other[, p])	Compute the minimum distance between points in the two hyperrectangles.
split(d, split)	Produce two hyperrectangles by splitting.
volume()	Total volume.

Rectangle.**max_distance_point**(*x*, *p=2.0*)

Return the maximum distance between input and points in the hyperrectangle.

Parameters **x** : array_like

Input array.

p : float, optional

Input.

Rectangle.**max_distance_rectangle**(*other*, *p=2.0*)

Compute the maximum distance between points in the two hyperrectangles.

Parameters **other** : hyperrectangle

Input.

p : float, optional

Input.

Rectangle.**min_distance_point**(*x*, *p=2.0*)

Return the minimum distance between input and points in the hyperrectangle.

Parameters **x** : array_like

Input.

p : float, optional

Input.

`Rectangle.`**`min_distance_rectangle`**(*other*, *p=2.0*)
Compute the minimum distance between points in the two hyperrectangles.

Parameters **other** : hyperrectangle
Input.
p : float
Input.

`Rectangle.`**`split`**(*d*, *split*)
Produce two hyperrectangles by splitting.

In general, if you need to compute maximum and minimum distances to the children, it can be done more efficiently by updating the maximum and minimum distances to the parent.

Parameters **d** : int
Axis to split hyperrectangle along.
split : float
Position along axis *d* to split at.

`Rectangle.`**`volume`**()
Total volume.

5.24.2 Delaunay Triangulation, Convex Hulls and Voronoi Diagrams

Delaunay(points[, furthest_site, ...])	Delaunay tesselation in N dimensions.
ConvexHull(points[, incremental, qhull_options])	Convex hulls in N dimensions.
Voronoi(points[, furthest_site, ...])	Voronoi diagrams in N dimensions.
SphericalVoronoi(points[, radius, center])	Voronoi diagrams on the surface of a sphere.
HalfspaceIntersection(halfspaces, interior_point)	Halfspace intersections in N dimensions.

class `scipy.spatial.`**`Delaunay`**(*points*, *furthest_site=False*, *incremental=False*, *qhull_options=None*)
Delaunay tesselation in N dimensions.

New in version 0.9.

Parameters **points** : ndarray of floats, shape (npoints, ndim)
Coordinates of points to triangulate
furthest_site : bool, optional
Whether to compute a furthest-site Delaunay triangulation. Default: False
New in version 0.12.0.
incremental : bool, optional
Allow adding new points incrementally. This takes up some additional resources.
qhull_options : str, optional
Additional options to pass to Qhull. See Qhull manual for details. Option "Qt" is always enabled. Default:"Qbb Qc Qz Qx Q12" for ndim > 4 and "Qbb Qc Qz Q12" otherwise. Incremental mode omits "Qz".
New in version 0.12.0.

Raises **QhullError**
Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.
ValueError
Raised if an incompatible array is given as input.

Notes

The tesselation is computed using the Qhull library Qhull library.

Note: Unless you pass in the Qhull option "QJ", Qhull does not guarantee that each input point appears as a vertex in the Delaunay triangulation. Omitted points are listed in the *coplanar* attribute.

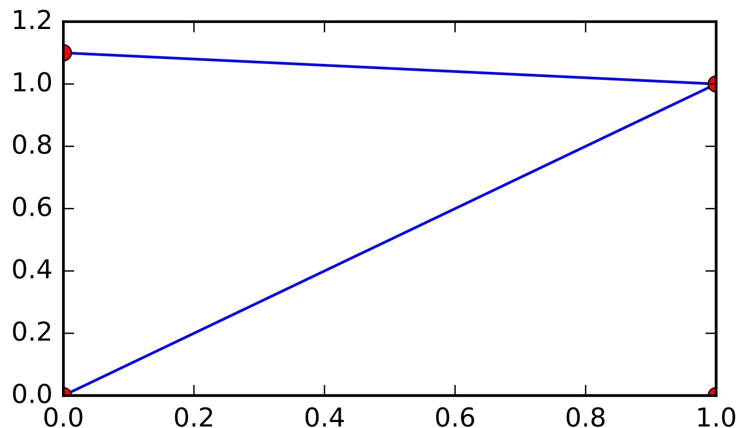
Examples

Triangulation of a set of points:

```
>>> points = np.array([[0, 0], [0, 1.1], [1, 0], [1, 1]])
>>> from scipy.spatial import Delaunay
>>> tri = Delaunay(points)
```

We can plot it:

```
>>> import matplotlib.pyplot as plt
>>> plt.triplot(points[:,0], points[:,1], tri.simplices.copy())
>>> plt.plot(points[:,0], points[:,1], 'o')
>>> plt.show()
```



Point indices and coordinates for the two triangles forming the triangulation:

```
>>> tri.simplices
array([[2, 3, 0],                 # may vary
       [3, 1, 0]], dtype=int32)
```

Note that depending on how rounding errors go, the simplices may be in a different order than above.

```
>>> points[tri.simplices]
array([[[ 1. ,  0. ],            # may vary
        [ 1. ,  1. ],
        [ 0. ,  0. ]],
       [[ 1. ,  1. ],
        [ 0. ,  1.1],
        [ 0. ,  0. ]]])
```

Triangle 0 is the only neighbor of triangle 1, and it's opposite to vertex 1 of triangle 1:

```
>>> tri.neighbors[1]
array([-1,  0, -1], dtype=int32)
>>> points[tri.simplices[1,1]]
array([ 0. ,  1.1])
```

We can find out which triangle points are in:

```
>>> p = np.array([(0.1, 0.2), (1.5, 0.5), (0.5, 1.05)])
>>> tri.find_simplex(p)
array([ 1, -1, 1], dtype=int32)
```

We can also compute barycentric coordinates in triangle 1 for these points:

```
>>> b = tri.transform[1,:2].dot(np.transpose(p - tri.transform[1,2]))
>>> np.c_[np.transpose(b), 1 - b.sum(axis=0)]
array([[ 0.1        ,  0.09090909,  0.80909091],
       [ 1.5        , -0.90909091,  0.40909091],
       [ 0.5        ,  0.5       ,  0.        ]])
```

The coordinates for the first point are all positive, meaning it is indeed inside the triangle. The third point is on a vertex, hence its null third coordinate.

Attributes

transform	Affine transform from `x` to the barycentric coordinates `c`.
vertex_to_simplex	Lookup array, from a vertex, to some simplex which it is a part of.
convex_hull	Vertices of facets forming the convex hull of the point set.
vertex_neighbor_vertices	Neighboring vertices of vertices.

`Delaunay.`**`transform`**

Affine transform from `x` to the barycentric coordinates `c`.

Type ndarray of double, shape (nsimplex, ndim+1, ndim)

This is defined by:

```
T c = x - r
```

At vertex `j`, `c_j = 1` and the other coordinates zero.

For simplex `i`, `transform[i,:ndim,:ndim]` contains inverse of the matrix `T`, and `transform[i,ndim,:]` contains the vector `r`.

`Delaunay.`**`vertex_to_simplex`**

Lookup array, from a vertex, to some simplex which it is a part of.

Type ndarray of int, shape (npoints,)

`Delaunay.`**`convex_hull`**

Vertices of facets forming the convex hull of the point set.

Type ndarray of int, shape (nfaces, ndim)

The array contains the indices of the points belonging to the (N-1)-dimensional facets that form the convex hull of the triangulation.

Note: Computing convex hulls via the Delaunay triangulation is inefficient and subject to increased numerical instability. Use `ConvexHull` instead.

`Delaunay.`**`vertex_neighbor_vertices`**

Neighboring vertices of vertices.

Tuple of two ndarrays of int: (indices, indptr). The indices of neighboring vertices of vertex *k* are `indptr[indices[k]:indices[k+1]]`.

points	(ndarray of double, shape (npoints, ndim)) Coordinates of input points.
simplices	(ndarray of ints, shape (nsimplex, ndim+1)) Indices of the points forming the simplices in the triangulation. For 2-D, the points are oriented counterclockwise.
neighbors	(ndarray of ints, shape (nsimplex, ndim+1)) Indices of neighbor simplices for each simplex. The kth neighbor is opposite to the kth vertex. For simplices at the boundary, -1 denotes no neighbor.
equations	(ndarray of double, shape (nsimplex, ndim+2)) [normal, offset] forming the hyperplane equation of the facet on the paraboloid (see Qhull documentation for more).
paraboloid_scale, paraboloid_shift	(float) Scale and shift for the extra paraboloid dimension (see Qhull documentation for more).
coplanar	(ndarray of int, shape (ncoplanar, 3)) Indices of coplanar points and the corresponding indices of the nearest facet and the nearest vertex. Coplanar points are input points which were *not* included in the triangulation due to numerical precision issues. If option "Qc" is not specified, this list is not computed. .. versionadded:: 0.12.0
vertices	Same as *simplices*, but deprecated.

Methods

add_points(points[, restart])	Process a set of additional new points.
close()	Finish incremental processing.
find_simplex(self, xi[, bruteforce, tol])	Find the simplices containing the given points.
lift_points(self, x)	Lift points to the Qhull paraboloid.
plane_distance(self, xi)	Compute hyperplane distances to the point *xi* from all simplices.

`Delaunay.`**`add_points`**(*points*, *restart=False*)

Process a set of additional new points.

Parameters **points** : ndarray

New points to add. The dimensionality should match that of the initial points.

restart : bool, optional

Whether to restart processing from scratch, rather than adding points incrementally.

Raises **QhullError**

Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

See also:

`close`

Notes

You need to specify `incremental=True` when constructing the object to be able to add points incrementally. Incremental addition of points is also not possible after `close` has been called.

`Delaunay.`**`close`**`()`

Finish incremental processing.

Call this to free resources taken up by Qhull, when using the incremental mode. After calling this, adding more points is no longer possible.

`Delaunay.`**`find_simplex`**`(`*self*, *xi*, *bruteforce=False*, *tol=None*`)`

Find the simplices containing the given points.

Parameters

tri : DelaunayInfo
: Delaunay triangulation

xi : ndarray of double, shape (..., ndim)
: Points to locate

bruteforce : bool, optional
: Whether to only perform a brute-force search

tol : float, optional
: Tolerance allowed in the inside-triangle check. Default is `100*eps`.

Returns

i : ndarray of int, same shape as *xi*
: Indices of simplices containing each point. Points outside the triangulation get the value -1.

Notes

This uses an algorithm adapted from Qhull's `qh_findbestfacet`, which makes use of the connection between a convex hull and a Delaunay triangulation. After finding the simplex closest to the point in N+1 dimensions, the algorithm falls back to directed search in N dimensions.

`Delaunay.`**`lift_points`**`(`*self*, *x*`)`

Lift points to the Qhull paraboloid.

`Delaunay.`**`plane_distance`**`(`*self*, *xi*`)`

Compute hyperplane distances to the point *xi* from all simplices.

class `scipy.spatial.`**`ConvexHull`**`(`*points*, *incremental=False*, *qhull_options=None*`)`

Convex hulls in N dimensions.

New in version 0.12.0.

Parameters

points : ndarray of floats, shape (npoints, ndim)
: Coordinates of points to construct a convex hull from

incremental : bool, optional
: Allow adding new points incrementally. This takes up some additional resources.

qhull_options : str, optional
: Additional options to pass to Qhull. See Qhull manual for details. (Default: "Qx" for ndim > 4 and "" otherwise) Option "Qt" is always enabled.

Raises

QhullError
: Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

ValueError
: Raised if an incompatible array is given as input.

Notes

The convex hull is computed using the Qhull library.

References

[Qhull]

Examples

Convex hull of a random set of points:

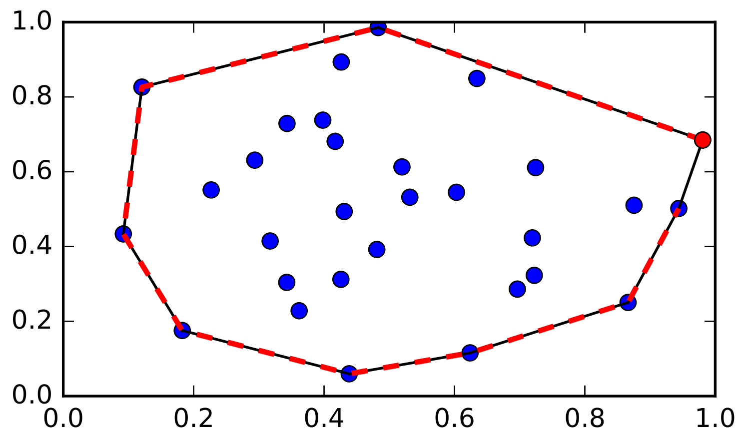
```
>>> from scipy.spatial import ConvexHull
>>> points = np.random.rand(30, 2)   # 30 random points in 2-D
>>> hull = ConvexHull(points)
```

Plot it:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(points[:,0], points[:,1], 'o')
>>> for simplex in hull.simplices:
...     plt.plot(points[simplex, 0], points[simplex, 1], 'k-')
```

We could also have directly used the vertices of the hull, which for 2-D are guaranteed to be in counterclockwise order:

```
>>> plt.plot(points[hull.vertices,0], points[hull.vertices,1], 'r--', lw=2)
>>> plt.plot(points[hull.vertices[0],0], points[hull.vertices[0],1], 'ro')
>>> plt.show()
```



Attributes

points	(ndarray of double, shape (npoints, ndim)) Coordinates of input points.
vertices	(ndarray of ints, shape (nvertices,)) Indices of points forming the vertices of the convex hull. For 2-D convex hulls, the vertices are in counterclockwise order. For other dimensions, they are in input order.
simplices	(ndarray of ints, shape (nfacet, ndim)) Indices of points forming the simplical facets of the convex hull.
neighbors	(ndarray of ints, shape (nfacet, ndim)) Indices of neighbor facets for each facet. The kth neighbor is opposite to the kth vertex. -1 denotes no neighbor.
equations	(ndarray of double, shape (nfacet, ndim+1)) [normal, offset] forming the hyperplane equation of the facet (see Qhull documentation for more).
coplanar	(ndarray of int, shape (ncoplanar, 3)) Indices of coplanar points and the corresponding indices of the nearest facets and nearest vertex indices. Coplanar points are input points which were *not* included in the triangulation due to numerical precision issues. If option "Qc" is not specified, this list is not computed.
area	(float) Area of the convex hull
volume	(float) Volume of the convex hull

Methods

`add_points`(points[, restart])	Process a set of additional new points.
`close`()	Finish incremental processing.

`ConvexHull.`**`add_points`**(*points*, *restart=False*)

Process a set of additional new points.

Parameters **points** : ndarray

New points to add. The dimensionality should match that of the initial points.

restart : bool, optional

Whether to restart processing from scratch, rather than adding points incrementally.

Raises **QhullError**

Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

See also:

`close`

Notes

You need to specify `incremental=True` when constructing the object to be able to add points incrementally. Incremental addition of points is also not possible after `close` has been called.

`ConvexHull.`**`close`**()

Finish incremental processing.

Call this to free resources taken up by Qhull, when using the incremental mode. After calling this, adding more points is no longer possible.

class `scipy.spatial.`**`Voronoi`**(*points*, *furthest_site=False*, *incremental=False*, *qhull_options=None*)

Voronoi diagrams in N dimensions.

New in version 0.12.0.

Parameters

points : ndarray of floats, shape (npoints, ndim)
: Coordinates of points to construct a convex hull from

furthest_site : bool, optional
: Whether to compute a furthest-site Voronoi diagram. Default: False

incremental : bool, optional
: Allow adding new points incrementally. This takes up some additional resources.

qhull_options : str, optional
: Additional options to pass to Qhull. See Qhull manual for details. (Default: "Qbb Qc Qz Qx" for ndim > 4 and "Qbb Qc Qz" otherwise. Incremental mode omits "Qz".)

Raises

QhullError
: Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

ValueError
: Raised if an incompatible array is given as input.

Notes

The Voronoi diagram is computed using the Qhull library.

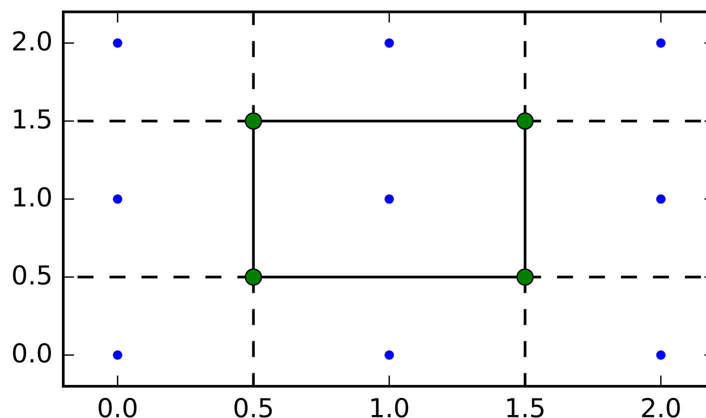
Examples

Voronoi diagram for a set of point:

```
>>> points = np.array([[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2],
...                    [2, 0], [2, 1], [2, 2]])
>>> from scipy.spatial import Voronoi, voronoi_plot_2d
>>> vor = Voronoi(points)
```

Plot it:

```
>>> import matplotlib.pyplot as plt
>>> voronoi_plot_2d(vor)
>>> plt.show()
```



The Voronoi vertices:

```
>>> vor.vertices
array([[ 0.5,  0.5],
       [ 1.5,  0.5],
       [ 0.5,  1.5],
       [ 1.5,  1.5]])
```

There is a single finite Voronoi region, and four finite Voronoi ridges:

```
>>> vor.regions
[[], [-1, 0], [-1, 1], [1, -1, 0], [3, -1, 2], [-1, 3], [-1, 2], [3, 2, 0, 1], [2,
↪ -1, 0], [3, -1, 1]]
>>> vor.ridge_vertices
[[-1, 0], [-1, 0], [-1, 1], [-1, 1], [0, 1], [-1, 3], [-1, 2], [2, 3], [-1, 3], [-
↪1, 2], [0, 2], [1, 3]]
```

The ridges are perpendicular between lines drawn between the following input points:

```
>>> vor.ridge_points
array([[0, 1],
       [0, 3],
       [6, 3],
       [6, 7],
       [3, 4],
       [5, 8],
       [5, 2],
       [5, 4],
       [8, 7],
       [2, 1],
       [4, 1],
       [4, 7]], dtype=int32)
```

Attributes

points	(ndarray of double, shape (npoints, ndim)) Coordinates of input points.
vertices	(ndarray of double, shape (nvertices, ndim)) Coordinates of the Voronoi vertices.
ridge_points	(ndarray of ints, shape `(nridges, 2)`) Indices of the points between which each Voronoi ridge lies.
ridge_vertices	(list of list of ints, shape `(nridges, *)`) Indices of the Voronoi vertices forming each Voronoi ridge.
regions	(list of list of ints, shape `(nregions, *)`) Indices of the Voronoi vertices forming each Voronoi region. -1 indicates vertex outside the Voronoi diagram.
point_region	(list of ints, shape (npoints)) Index of the Voronoi region for each input point. If qhull option “Qc” was not specified, the list will contain -1 for points that are not associated with a Voronoi region.

Methods

add_points(points[, restart])	Process a set of additional new points.
close()	Finish incremental processing.

`Voronoi.`**add_points**(*points*, *restart=False*)

Process a set of additional new points.

Parameters **points** : ndarray

New points to add. The dimensionality should match that of the initial

points.

restart : bool, optional
Whether to restart processing from scratch, rather than adding points incrementally.

Raises **QhullError**
Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

See also:

`close`

Notes

You need to specify `incremental=True` when constructing the object to be able to add points incrementally. Incremental addition of points is also not possible after `close` has been called.

`Voronoi.`**`close`**`()`
Finish incremental processing.

Call this to free resources taken up by Qhull, when using the incremental mode. After calling this, adding more points is no longer possible.

class `scipy.spatial.`**`SphericalVoronoi`**(*points*, *radius=None*, *center=None*)
Voronoi diagrams on the surface of a sphere.

New in version 0.18.0.

Parameters **points** : ndarray of floats, shape (npoints, 3)
Coordinates of points to construct a spherical Voronoi diagram from

radius : float, optional
Radius of the sphere (Default: 1)

center : ndarray of floats, shape (3,)
Center of sphere (Default: origin)

See also:

`Voronoi` Conventional Voronoi diagrams in N dimensions.

Notes

The spherical Voronoi diagram algorithm proceeds as follows. The Convex Hull of the input points (generators) is calculated, and is equivalent to their Delaunay triangulation on the surface of the sphere *[Caroli]*. A 3D Delaunay tetrahedralization is obtained by including the origin of the coordinate system as the fourth vertex of each simplex of the Convex Hull. The circumcenters of all tetrahedra in the system are calculated and projected to the surface of the sphere, producing the Voronoi vertices. The Delaunay tetrahedralization neighbour information is then used to order the Voronoi region vertices around each generator. The latter approach is substantially less sensitive to floating point issues than angle-based methods of Voronoi region vertex sorting.

The surface area of spherical polygons is calculated by decomposing them into triangles and using L'Huilier's Theorem to calculate the spherical excess of each triangle *[Weisstein]*. The sum of the spherical excesses is multiplied by the square of the sphere radius to obtain the surface area of the spherical polygon. For nearly-degenerate spherical polygons an area of approximately 0 is returned by default, rather than attempting the unstable calculation.

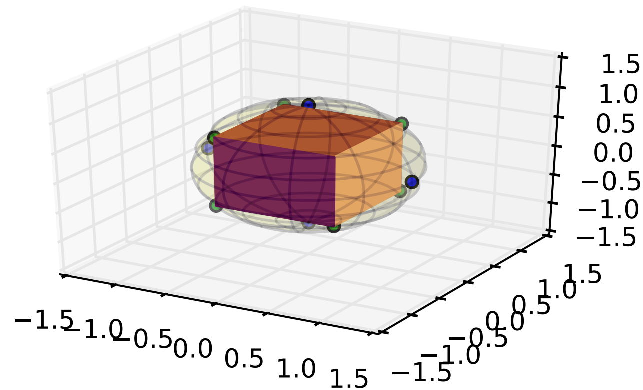
Empirical assessment of spherical Voronoi algorithm performance suggests quadratic time complexity (loglinear is optimal, but algorithms are more challenging to implement). The reconstitution of the surface area of the sphere, measured as the sum of the surface areas of all Voronoi regions, is closest to 100 % for larger (>> 10) numbers of generators.

References

[Caroli], [Weisstein]

Examples

```
>>> from matplotlib import colors
>>> from mpl_toolkits.mplot3d.art3d import Poly3DCollection
>>> import matplotlib.pyplot as plt
>>> from scipy.spatial import SphericalVoronoi
>>> from mpl_toolkits.mplot3d import proj3d
>>> # set input data
>>> points = np.array([[0, 0, 1], [0, 0, -1], [1, 0, 0],
...                    [0, 1, 0], [0, -1, 0], [-1, 0, 0], ])
>>> center = np.array([0, 0, 0])
>>> radius = 1
>>> # calculate spherical Voronoi diagram
>>> sv = SphericalVoronoi(points, radius, center)
>>> # sort vertices (optional, helpful for plotting)
>>> sv.sort_vertices_of_regions()
>>> # generate plot
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> # plot the unit sphere for reference (optional)
>>> u = np.linspace(0, 2 * np.pi, 100)
>>> v = np.linspace(0, np.pi, 100)
>>> x = np.outer(np.cos(u), np.sin(v))
>>> y = np.outer(np.sin(u), np.sin(v))
>>> z = np.outer(np.ones(np.size(u)), np.cos(v))
>>> ax.plot_surface(x, y, z, color='y', alpha=0.1)
>>> # plot generator points
>>> ax.scatter(points[:, 0], points[:, 1], points[:, 2], c='b')
>>> # plot Voronoi vertices
>>> ax.scatter(sv.vertices[:, 0], sv.vertices[:, 1], sv.vertices[:, 2],
...                    c='g')
>>> # indicate Voronoi regions (as Euclidean polygons)
>>> for region in sv.regions:
...    random_color = colors.rgb2hex(np.random.rand(3))
...    polygon = Poly3DCollection([sv.vertices[region]], alpha=1.0)
...    polygon.set_color(random_color)
...    ax.add_collection3d(polygon)
>>> plt.show()
```



Attributes

points	(double array of shape (npoints, 3)) the points in 3D to generate the Voronoi diagram from
radius	(double) radius of the sphere Default: None (forces estimation, which is less precise)
center	(double array of shape (3,)) center of the sphere Default: None (assumes sphere is centered at origin)
vertices	(double array of shape (nvertices, 3)) Voronoi vertices corresponding to points
regions	(list of list of integers of shape (npoints, _)) the n-th entry is a list consisting of the indices of the vertices belonging to the n-th point in points

Methods

sort_vertices_of_regions()	For each region in regions, it sorts the indices of the Voronoi vertices such that the resulting points are in a clockwise or counterclockwise order around the generator point.

SphericalVoronoi.**sort_vertices_of_regions**()

For each region in regions, it sorts the indices of the Voronoi vertices such that the resulting points are in a clockwise or counterclockwise order around the generator point.

This is done as follows: Recall that the n-th region in regions surrounds the n-th generator in points and that the k-th Voronoi vertex in vertices is the projected circumcenter of the tetrahedron obtained by the k-th triangle in _tri.simplices (and the origin). For each region n, we choose the first triangle (=Voronoi vertex) in _tri.simplices and a vertex of that triangle not equal to the center n. These determine a unique neighbor of that triangle, which is then chosen as the second triangle. The second triangle will have a unique vertex not equal to the current vertex or the center. This determines a unique neighbor of the second triangle, which is then chosen as the third triangle and so forth. We proceed through all the triangles (=Voronoi vertices) belonging to the generator in points and obtain a sorted version of the vertices of its surrounding region.

class scipy.spatial.**HalfspaceIntersection**(*halfspaces*, *interior_point*, *incremental=False*, *qhull_options=None*)

Halfspace intersections in N dimensions.

New in version 0.19.0.

Parameters

halfspaces : ndarray of floats, shape (nineq, ndim+1)
: Stacked Inequalities of the form Ax + b <= 0 in format [A; b]

interior_point : ndarray of floats, shape (ndim,)
: Point clearly inside the region defined by halfspaces. Also called a feasible point, it can be obtained by linear programming.

incremental : bool, optional
: Allow adding new halfspaces incrementally. This takes up some additional resources.

qhull_options : str, optional
: Additional options to pass to Qhull. See Qhull manual for details. (Default: "Qx" for ndim > 4 and "" otherwise) Option "H" is always enabled.

Raises

QhullError
: Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

ValueError
: Raised if an incompatible array is given as input.

Notes

The intersections are computed using the Qhull library. This reproduces the "qhalf" functionality of Qhull.

References

[Qhull], *[R350]*

Examples

Halfspace intersection of planes forming some polygon

```
>>> from scipy.spatial import HalfspaceIntersection
>>> import numpy as np
>>> halfspaces = np.array([[-1, 0., 0.],
...                        [0., -1., 0.],
...                        [2., 1., -4.],
...                        [-0.5, 1., -2.]])
>>> feasible_point = np.array([0.5, 0.5])
>>> hs = HalfspaceIntersection(halfspaces, feasible_point)
```

Plot halfspaces as filled regions and intersection points:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot('111', aspect='equal')
>>> xlim, ylim = (-1, 3), (-1, 3)
>>> ax.set_xlim(xlim)
>>> ax.set_ylim(ylim)
>>> x = np.linspace(-1, 3, 100)
>>> symbols = ['-', '+', 'x', '*']
>>> signs = [0, 0, -1, -1]
>>> fmt = {"color": None, "edgecolor": "b", "alpha": 0.5}
>>> for h, sym, sign in zip(halfspaces, symbols, signs):
...     hlist = h.tolist()
...     fmt["hatch"] = sym
...     if h[1]== 0:
...         ax.axvline(-h[2]/h[0], label='{}x+{}y+{}=0'.format(*hlist))
```

```
...             xi = np.linspace(xlim[sign], -h[2]/h[0], 100)
...             ax.fill_between(xi, ylim[0], ylim[1], **fmt)
...         else:
...             ax.plot(x, (-h[2]-h[0]*x)/h[1], label='{}x+{}y+{}=0'.format(*hlist))
...             ax.fill_between(x, (-h[2]-h[0]*x)/h[1], ylim[sign], **fmt)
>>> x, y = zip(*hs.intersections)
>>> ax.plot(x, y, 'o', markersize=8)
```

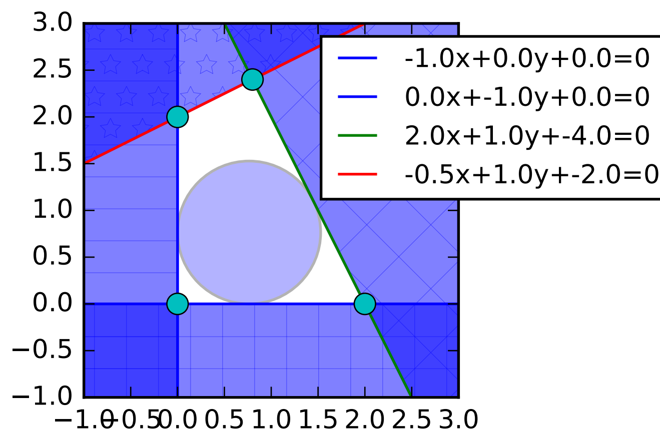
By default, qhull does not provide with a way to compute an interior point. This can easily be computed using linear programming. Considering halfspaces of the form $Ax + b \leq 0$, solving the linear program:

$$max\ y$$
$$s.t. Ax + y||A_i|| \leq -b$$

With A_i being the rows of A, i.e. the normals to each plane.

Will yield a point x that is furthest inside the convex polyhedron. To be precise, it is the center of the largest hypersphere of radius y inscribed in the polyhedron. This point is called the Chebyshev center of the polyhedron (see *[R350]* 4.3.1, pp148-149). The equations outputted by Qhull are always normalized.

```
>>> from scipy.optimize import linprog
>>> from matplotlib.patches import Circle
>>> norm_vector = np.reshape(np.linalg.norm(halfspaces[:, :-1], axis=1),
...     (halfspaces.shape[0], 1))
>>> c = np.zeros((halfspaces.shape[1],))
>>> c[-1] = -1
>>> A = np.hstack((halfspaces[:, :-1], norm_vector))
>>> b = - halfspaces[:, -1:]
>>> res = linprog(c, A_ub=A, b_ub=b)
>>> x = res.x[:-1]
>>> y = res.x[-1]
>>> circle = Circle(x, radius=y, alpha=0.3)
>>> ax.add_patch(circle)
>>> plt.legend(bbox_to_anchor=(1.6, 1.0))
>>> plt.show()
```



Attributes

halfspaces	(ndarray of double, shape (nineq, ndim+1)) Input halfspaces.
interior_point :ndarray of floats, shape (ndim,)	Input interior point.
intersections	(ndarray of double, shape (ninter, ndim)) Intersections of all halfspaces.
dual_points	(ndarray of double, shape (nineq, ndim)) Dual points of the input halfspaces.
dual_facets	(list of lists of ints) Indices of points forming the (non necessarily simplicial) facets of the dual convex hull.
dual_vertices	(ndarray of ints, shape (nvertices,)) Indices of halfspaces forming the vertices of the dual convex hull. For 2-D convex hulls, the vertices are in counterclockwise order. For other dimensions, they are in input order.
dual_equations	(ndarray of double, shape (nfacet, ndim+1)) [normal, offset] forming the hyperplane equation of the dual facet (see Qhull documentation for more).
dual_area	(float) Area of the dual convex hull
dual_volume	(float) Volume of the dual convex hull

Methods

`add_halfspaces`(halfspaces[, restart])	Process a set of additional new halfspaces.
`close`()	Finish incremental processing.

`HalfspaceIntersection.`**`add_halfspaces`**(*halfspaces*, *restart=False*)

Process a set of additional new halfspaces.

Parameters **halfspaces** : ndarray

New halfspaces to add. The dimensionality should match that of the initial halfspaces.

restart : bool, optional

Whether to restart processing from scratch, rather than adding halfspaces incrementally.

Raises **QhullError**

Raised when Qhull encounters an error condition, such as geometrical degeneracy when options to resolve are not enabled.

See also:

`close`

Notes

You need to specify `incremental=True` when constructing the object to be able to add halfspaces incrementally. Incremental addition of halfspaces is also not possible after `close` has been called.

`HalfspaceIntersection.`**`close`**()

Finish incremental processing.

Call this to free resources taken up by Qhull, when using the incremental mode. After calling this, adding more points is no longer possible.

5.24.3 Plotting Helpers

`delaunay_plot_2d`(tri[, ax])	Plot the given Delaunay triangulation in 2-D

Continued on next page

Table 5.224 – continued from previous page

convex_hull_plot_2d(hull[, ax])	Plot the given convex hull diagram in 2-D
voronoi_plot_2d(vor[, ax])	Plot the given Voronoi diagram in 2-D

scipy.spatial.**delaunay_plot_2d**(*tri*, *ax=None*)

Plot the given Delaunay triangulation in 2-D

Parameters **tri** : scipy.spatial.Delaunay instance

Triangulation to plot

ax : matplotlib.axes.Axes instance, optional

Axes to plot on

Returns **fig** : matplotlib.figure.Figure instance

Figure for the plot

See also:

Delaunay, matplotlib.pyplot.triplot

Notes

Requires Matplotlib.

scipy.spatial.**convex_hull_plot_2d**(*hull*, *ax=None*)

Plot the given convex hull diagram in 2-D

Parameters **hull** : scipy.spatial.ConvexHull instance

Convex hull to plot

ax : matplotlib.axes.Axes instance, optional

Axes to plot on

Returns **fig** : matplotlib.figure.Figure instance

Figure for the plot

See also:

ConvexHull

Notes

Requires Matplotlib.

scipy.spatial.**voronoi_plot_2d**(*vor*, *ax=None*, ***kw*)

Plot the given Voronoi diagram in 2-D

Parameters **vor** : scipy.spatial.Voronoi instance

Diagram to plot

ax : matplotlib.axes.Axes instance, optional

Axes to plot on

show_points: bool, optional

Add the Voronoi points to the plot.

show_vertices : bool, optional

Add the Voronoi vertices to the plot.

line_colors : string, optional

Specifies the line color for polygon boundaries

line_width : float, optional

Specifies the line width for polygon boundaries

line_alpha: float, optional

Specifies the line alpha for polygon boundaries

Returns **fig** : matplotlib.figure.Figure instance

Figure for the plot

See also:

Voronoi

Notes

Requires Matplotlib.

See also:

Tutorial

5.24.4 Simplex representation

The simplices (triangles, tetrahedra, ...) appearing in the Delaunay tesselation (N-dim simplices), convex hull facets, and Voronoi ridges (N-1 dim simplices) are represented in the following scheme:

```
tess = Delaunay(points)
hull = ConvexHull(points)
voro = Voronoi(points)

# coordinates of the j-th vertex of the i-th simplex
tess.points[tess.simplices[i, j], :]        # tesselation element
hull.points[hull.simplices[i, j], :]        # convex hull facet
voro.vertices[voro.ridge_vertices[i, j], :] # ridge between Voronoi cells
```

For Delaunay triangulations and convex hulls, the neighborhood structure of the simplices satisfies the condition:

`tess.neighbors[i,j]` is the neighboring simplex of the i-th simplex, opposite to the j-vertex. It is -1 in case of no neighbor.

Convex hull facets also define a hyperplane equation:

```
(hull.equations[i,:-1] * coord).sum() + hull.equations[i,-1] == 0
```

Similar hyperplane equations for the Delaunay triangulation correspond to the convex hull facets on the corresponding N+1 dimensional paraboloid.

The Delaunay triangulation objects offer a method for locating the simplex containing a given point, and barycentric coordinate computations.

Functions

tsearch(tri, xi)	Find simplices containing the given points.
distance_matrix(x, y[, p, threshold])	Compute the distance matrix.
minkowski_distance(x, y[, p])	Compute the L**p distance between two arrays.
minkowski_distance_p(x, y[, p])	Compute the p-th power of the L**p distance between two arrays.
procrustes(data1, data2)	Procrustes analysis, a similarity test for two data sets.

scipy.spatial.**tsearch**(*tri*, *xi*)

Find simplices containing the given points. This function does the same thing as *Delaunay.find_simplex*.

New in version 0.9.

See also:

Delaunay.find_simplex

scipy.spatial.**distance_matrix**(*x*, *y*, *p=2*, *threshold=1000000*)
Compute the distance matrix.

Returns the matrix of all pair-wise distances.

Parameters

x : (M, K) array_like
Matrix of M vectors in K dimensions.

y : (N, K) array_like
Matrix of N vectors in K dimensions.

p : float, 1 <= p <= infinity
Which Minkowski p-norm to use.

threshold : positive int
If `M * N * K` > *threshold*, algorithm uses a Python loop instead of large temporary arrays.

Returns

result : (M, N) ndarray
Matrix containing the distance from every vector in *x* to every vector in *y*.

Examples

```
>>> from scipy.spatial import distance_matrix
>>> distance_matrix([[0,0],[0,1]], [[1,0],[1,1]])
array([[ 1.        ,  1.41421356],
       [ 1.41421356,  1.        ]])
```

scipy.spatial.**minkowski_distance**(*x*, *y*, *p=2*)
Compute the L**p distance between two arrays.

Parameters

x : (M, K) array_like
Input array.

y : (N, K) array_like
Input array.

p : float, 1 <= p <= infinity
Which Minkowski p-norm to use.

Examples

```
>>> from scipy.spatial import minkowski_distance
>>> minkowski_distance([[0,0],[0,0]], [[1,1],[0,1]])
array([ 1.41421356,  1.        ])
```

scipy.spatial.**minkowski_distance_p**(*x*, *y*, *p=2*)
Compute the p-th power of the L**p distance between two arrays.

For efficiency, this function computes the L**p distance but does not extract the pth root. If *p* is 1 or infinity, this is equal to the actual L**p distance.

Parameters

x : (M, K) array_like
Input array.

y : (N, K) array_like
Input array.

p : float, 1 <= p <= infinity
Which Minkowski p-norm to use.

Examples

```
>>> from scipy.spatial import minkowski_distance_p
>>> minkowski_distance_p([[0,0],[0,0]], [[1,1],[0,1]])
array([2, 1])
```

`scipy.spatial.`**`procrustes`**`(`*data1*, *data2*`)`

Procrustes analysis, a similarity test for two data sets.

Each input matrix is a set of points or vectors (the rows of the matrix). The dimension of the space is the number of columns of each matrix. Given two identically sized matrices, procrustes standardizes both such that:

- $tr(AA^T) = 1$.
- Both sets of points are centered around the origin.

Procrustes (*[R365]*, *[R366]*) then applies the optimal transform to the second matrix (including scaling/dilation, rotations, and reflections) to minimize $M^2 = \sum(data1 - data2)^2$, or the sum of the squares of the pointwise differences between the two input datasets.

This function was not designed to handle datasets with different numbers of datapoints (rows). If two data sets have different dimensionality (different number of columns), simply add columns of zeros to the smaller of the two.

Parameters

data1 : array_like
: Matrix, n rows represent points in k (columns) space *data1* is the reference data, after it is standardised, the data from *data2* will be transformed to fit the pattern in *data1* (must have >1 unique points).

data2 : array_like
: n rows of data in k space to be fit to *data1*. Must be the same shape `(numrows, numcols)` as data1 (must have >1 unique points).

Returns

mtx1 : array_like
: A standardized version of *data1*.

mtx2 : array_like
: The orientation of *data2* that best fits *data1*. Centered, but not necessarily $tr(AA^T) = 1$.

disparity : float
: M^2 as defined above.

Raises

ValueError
: If the input arrays are not two-dimensional. If the shape of the input arrays is different. If the input arrays have zero columns or zero rows.

See also:

`scipy.linalg.orthogonal_procrustes`

`scipy.spatial.distance.directed_hausdorff`
: Another similarity test for two data sets

Notes

- The disparity should not depend on the order of the input matrices, but the output matrices will, as only the first output matrix is guaranteed to be scaled such that $tr(AA^T) = 1$.
- Duplicate data points are generally ok, duplicating a data point will increase its effect on the procrustes fit.
- The disparity scales as the number of points per input matrix.

References

[R365], *[R366]*

Examples

```
>>> from scipy.spatial import procrustes
```

The matrix b is a rotated, shifted, scaled and mirrored version of a here:

```
>>> a = np.array([[1, 3], [1, 2], [1, 1], [2, 1]], 'd')
>>> b = np.array([[4, -2], [4, -4], [4, -6], [2, -6]], 'd')
>>> mtx1, mtx2, disparity = procrustes(a, b)
>>> round(disparity)
0.0
```

5.25 Distance computations (scipy.spatial.distance)

5.25.1 Function Reference

Distance matrix computation from a collection of raw observation vectors stored in a rectangular array.

pdist(X[, metric, p, w, V, VI])	Pairwise distances between observations in n-dimensional space.
cdist(XA, XB[, metric, p, V, VI, w])	Computes distance between each pair of the two collections of inputs.
squareform(X[, force, checks])	Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.
directed_hausdorff(u, v[, seed])	Computes the directed Hausdorff distance between two N-D arrays.

scipy.spatial.distance.**pdist**(*X*, *metric='euclidean'*, *p=None*, *w=None*, *V=None*, *VI=None*)

Pairwise distances between observations in n-dimensional space.

See Notes for common calling conventions.

Parameters

X : ndarray

An m by n array of m original observations in an n-dimensional space.

metric : str or function, optional

The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.

p : double, optional

The p-norm to apply Only for Minkowski, weighted and unweighted. Default: 2.

w : ndarray, optional

The weight vector. Only for weighted Minkowski. Mandatory

V : ndarray, optional

The variance vector Only for standardized Euclidean. Default: var(X, axis=0, ddof=1)

VI : ndarray, optional

The inverse of the covariance matrix Only for Mahalanobis. Default: inv(cov(X.T)).T

Returns

Y : ndarray

Returns a condensed distance matrix Y. For each i and j (where $i < j < m$),where m is the number of original observations. The metric `dist(u=X[i], v=X[j])` is computed and stored in entry `ij`.

See also:

`squareform`converts between condensed distance matrices and square distance matrices.

Notes

See `squareform` for information on how to calculate the index of this entry or to convert the condensed distance matrix to a redundant square matrix.

The following are common calling conventions.

1. `Y = pdist(X, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = pdist(X, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u - v||_p$ (p-norm) where $p \geq 1$.

3. `Y = pdist(X, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = pdist(X, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

   $$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}$$

   V is the variance vector; V[i] is the variance computed over all the i'th components of the points. If not passed, it is automatically computed.

5. `Y = pdist(X, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u - v||_2^2$ between the vectors.

6. `Y = pdist(X, 'cosine')`

   Computes the cosine distance between vectors u and v,

   $$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

   where $|| * ||_2$ is the 2-norm of its argument `*`, and $u \cdot v$ is the dot product of `u` and `v`.

7. `Y = pdist(X, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

   $$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

   where $\bar{v}$ is the mean of the elements of vector v, and $x \cdot y$ is the dot product of x and y.

8. `Y = pdist(X, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = pdist(X, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree.

10. `Y = pdist(X, 'chebyshev')`

Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

$$d(u,v) = \max_i |u_i - v_i|$$

11. `Y = pdist(X, 'canberra')`

Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

$$d(u,v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}$$

12. `Y = pdist(X, 'braycurtis')`

Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

$$d(u,v) = \frac{\sum_i |u_i - v_i|}{\sum_i |u_i + v_i|}$$

13. `Y = pdist(X, 'mahalanobis', VI=None)`

Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $\sqrt{(u-v)(1/V)(u-v)^T}$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

14. `Y = pdist(X, 'yule')`

Computes the Yule distance between each pair of boolean vectors. (see yule function documentation)

15. `Y = pdist(X, 'matching')`

Synonym for 'hamming'.

16. `Y = pdist(X, 'dice')`

Computes the Dice distance between each pair of boolean vectors. (see dice function documentation)

17. `Y = pdist(X, 'kulsinski')`

Computes the Kulsinski distance between each pair of boolean vectors. (see kulsinski function documentation)

18. `Y = pdist(X, 'rogerstanimoto')`

Computes the Rogers-Tanimoto distance between each pair of boolean vectors. (see rogerstanimoto function documentation)

19. `Y = pdist(X, 'russellrao')`

Computes the Russell-Rao distance between each pair of boolean vectors. (see russellrao function documentation)

20. `Y = pdist(X, 'sokalmichener')`

Computes the Sokal-Michener distance between each pair of boolean vectors. (see sokalmichener function documentation)

21. `Y = pdist(X, 'sokalsneath')`

    Computes the Sokal-Sneath distance between each pair of boolean vectors. (see sokalsneath function documentation)

22. `Y = pdist(X, 'wminkowski')`

    Computes the weighted Minkowski distance between each pair of vectors. (see wminkowski function documentation)

23. `Y = pdist(X, f)`

    Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

```
dm = pdist(X, lambda u, v: np.sqrt(((u-v)**2).sum()))
```

Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

```
dm = pdist(X, sokalsneath)
```

would calculate the pair-wise distances between the vectors in X using the Python function sokalsneath. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax.:

```
dm = pdist(X, 'sokalsneath')
```

`scipy.spatial.distance.`**`cdist`**(*XA*, *XB*, *metric='euclidean'*, *p=None*, *V=None*, *VI=None*, *w=None*)

Computes distance between each pair of the two collections of inputs.

See Notes for common calling conventions.

Parameters **XA** : ndarray

An m_A by n array of m_A original observations in an n-dimensional space. Inputs are converted to float type.

XB : ndarray

An m_B by n array of m_B original observations in an n-dimensional space. Inputs are converted to float type.

metric : str or callable, optional

The distance metric to use. If a string, the distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

p : double, optional

The p-norm to apply Only for Minkowski, weighted and unweighted. Default: 2.

w : ndarray, optional

The weight vector. Only for weighted Minkowski. Mandatory

V : ndarray, optional

The variance vector Only for standardized Euclidean. Default: var(vstack([XA, XB]), axis=0, ddof=1)

VI : ndarray, optional

The inverse of the covariance matrix Only for Mahalanobis. Default: inv(cov(vstack([XA, XB]).T)).T

Returns **Y** : ndarray

A m_A by m_B distance matrix is returned. For each i and j, the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the ij th entry.

Raises **ValueError**

An exception is thrown if *XA* and *XB* do not have the same number of columns.

Notes

The following are common calling conventions:

1. `Y = cdist(XA, XB, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = cdist(XA, XB, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = cdist(XA, XB, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = cdist(XA, XB, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

   $$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}.$$

   V is the variance vector; V[i] is the variance computed over all the i'th components of the points. If not passed, it is automatically computed.

5. `Y = cdist(XA, XB, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = cdist(XA, XB, 'cosine')`

   Computes the cosine distance between vectors u and v,

   $$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

   where $||*||_2$ is the 2-norm of its argument `*`, and $u \cdot v$ is the dot product of u and v.

7. `Y = cdist(XA, XB, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

   $$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

   where $\bar{v}$ is the mean of the elements of vector v, and $x \cdot y$ is the dot product of x and y.

8. `Y = cdist(XA, XB, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = cdist(XA, XB, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree where at least one of them is non-zero.

10. `Y = cdist(XA, XB, 'chebyshev')`

    Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

    $$d(u, v) = \max_i |u_i - v_i|.$$

11. `Y = cdist(XA, XB, 'canberra')`

    Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

$$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}.$$

12. `Y = cdist(XA, XB, 'braycurtis')`

    Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

$$d(u, v) = \frac{\sum_i (|u_i - v_i|)}{\sum_i (|u_i + v_i|)}$$

13. `Y = cdist(XA, XB, 'mahalanobis', VI=None)`

    Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $\sqrt{(u - v)(1/V)(u - v)^T}$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

14. `Y = cdist(XA, XB, 'yule')`

    Computes the Yule distance between the boolean vectors. (see *yule* function documentation)

15. `Y = cdist(XA, XB, 'matching')`

    Synonym for 'hamming'.

16. `Y = cdist(XA, XB, 'dice')`

    Computes the Dice distance between the boolean vectors. (see *dice* function documentation)

17. `Y = cdist(XA, XB, 'kulsinski')`

    Computes the Kulsinski distance between the boolean vectors. (see *kulsinski* function documentation)

18. `Y = cdist(XA, XB, 'rogerstanimoto')`

    Computes the Rogers-Tanimoto distance between the boolean vectors. (see *rogerstanimoto* function documentation)

19. `Y = cdist(XA, XB, 'russellrao')`

    Computes the Russell-Rao distance between the boolean vectors. (see *russellrao* function documentation)

20. `Y = cdist(XA, XB, 'sokalmichener')`

    Computes the Sokal-Michener distance between the boolean vectors. (see *sokalmichener* function documentation)

21. `Y = cdist(XA, XB, 'sokalsneath')`

    Computes the Sokal-Sneath distance between the vectors. (see *sokalsneath* function documentation)

22. `Y = cdist(XA, XB, 'wminkowski')`

Computes the weighted Minkowski distance between the vectors. (see *wminkowski* function documentation)

23. `Y = cdist(XA, XB, f)`

Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

```
dm = cdist(XA, XB, lambda u, v: np.sqrt(((u-v)**2).sum()))
```

Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

```
dm = cdist(XA, XB, sokalsneath)
```

would calculate the pair-wise distances between the vectors in X using the Python function *sokalsneath*. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax:

```
dm = cdist(XA, XB, 'sokalsneath')
```

Examples

Find the Euclidean distances between four 2-D coordinates:

```
>>> from scipy.spatial import distance
>>> coords = [(35.0456, -85.2672),
...           (35.1174, -89.9711),
...           (35.9728, -83.9422),
...           (36.1667, -86.7833)]
>>> distance.cdist(coords, coords, 'euclidean')
array([[ 0.    ,  4.7044,  1.6172,  1.8856],
       [ 4.7044,  0.    ,  6.0893,  3.3561],
       [ 1.6172,  6.0893,  0.    ,  2.8477],
       [ 1.8856,  3.3561,  2.8477,  0.    ]])
```

Find the Manhattan distance from a 3-D point to the corners of the unit cube:

```
>>> a = np.array([[0, 0, 0],
...               [0, 0, 1],
...               [0, 1, 0],
...               [0, 1, 1],
...               [1, 0, 0],
...               [1, 0, 1],
...               [1, 1, 0],
...               [1, 1, 1]])
>>> b = np.array([[ 0.1,  0.2,  0.4]])
>>> distance.cdist(a, b, 'cityblock')
array([[ 0.7],
       [ 0.9],
       [ 1.3],
       [ 1.5],
       [ 1.5],
       [ 1.7],
       [ 2.1],
       [ 2.3]])
```

`scipy.spatial.distance.`**`squareform`**(*X*, *force='no'*, *checks=True*)

Converts a vector-form distance vector to a square-form distance matrix, and vice-versa.

Parameters **X** : ndarray

Either a condensed or redundant distance matrix.

force : str, optional

As with MATLAB(TM), if force is equal to `'tovector'` or `'tomatrix'`, the input will be treated as a distance matrix or distance vector respectively.

checks : bool, optional

If set to False, no checks will be made for matrix symmetry nor zero diagonals. This is useful if it is known that `X - X.T1` is small and `diag(X)` is close to zero. These values are ignored any way so they do not disrupt the squareform transformation.

Returns **Y** : ndarray

If a condensed distance matrix is passed, a redundant one is returned, or if a redundant one is passed, a condensed distance matrix is returned.

Notes

1.v = squareform(X)

Given a square d-by-d symmetric distance matrix X, `v = squareform(X)` returns a `d * (d-1) / 2` (or $\binom{n}{2}$) sized vector v.

$v[\binom{n}{2} - \binom{n-i}{2} + (j-i-1)]$ is the distance between points i and j. If X is non-square or asymmetric, an error is returned.

2.X = squareform(v)

Given a `d*(d-1)/2` sized v for some integer `d >= 2` encoding distances as described, `X = squareform(v)` returns a d by d distance matrix X. The `X[i, j]` and `X[j, i]` values are set to $v[\binom{n}{2} - \binom{n-i}{2} + (j-i-1)]$ and all diagonal elements are zero.

In Scipy 0.19.0, `squareform` stopped casting all input types to float64, and started returning arrays of the same dtype as the input.

`scipy.spatial.distance.`**`directed_hausdorff`**(*u*, *v*, *seed=0*)

Computes the directed Hausdorff distance between two N-D arrays.

Distances between pairs are calculated using a Euclidean metric.

Parameters **u** : (M,N) ndarray

Input array.

v : (O,N) ndarray

Input array.

seed : int or None

Local `np.random.RandomState` seed. Default is 0, a random shuffling of u and v that guarantees reproducibility.

Returns **d** : double

The directed Hausdorff distance between arrays *u* and *v*,

index_1 : int

index of point contributing to Hausdorff pair in *u*

index_2 : int

index of point contributing to Hausdorff pair in *v*

See also:

`scipy.spatial.procrustes`

Another similarity test for two data sets

Notes

Uses the early break technique and the random sampling approach described by *[R364]*. Although worst-case performance is `O(m * o)` (as with the brute force algorithm), this is unlikely in practice as the input data would have to require the algorithm to explore every single point interaction, and after the algorithm shuffles the input points at that. The best case performance is O(m), which is satisfied by selecting an inner loop distance that is less than cmax and leads to an early break as often as possible. The authors have formally shown that the average runtime is closer to O(m).

New in version 0.19.0.

References

[R364]

Examples

Find the directed Hausdorff distance between two 2-D arrays of coordinates:

```
>>> from scipy.spatial.distance import directed_hausdorff
>>> u = np.array([(1.0, 0.0),
...               (0.0, 1.0),
...               (-1.0, 0.0),
...               (0.0, -1.0)])
>>> v = np.array([(2.0, 0.0),
...               (0.0, 2.0),
...               (-2.0, 0.0),
...               (0.0, -4.0)])
```

```
>>> directed_hausdorff(u, v)[0]
2.23606797749979
>>> directed_hausdorff(v, u)[0]
3.0
```

Find the general (symmetric) Hausdorff distance between two 2-D arrays of coordinates:

```
>>> max(directed_hausdorff(u, v)[0], directed_hausdorff(v, u)[0])
3.0
```

Find the indices of the points that generate the Hausdorff distance (the Hausdorff pair):

```
>>> directed_hausdorff(v, u)[1:]
(3, 3)
```

Predicates for checking the validity of distance matrices, both condensed and redundant. Also contained in this module are functions for computing the number of observations in a distance matrix.

`is_valid_dm`(D[, tol, throw, name, warning])	Returns True if input array is a valid distance matrix.
`is_valid_y`(y[, warning, throw, name])	Returns True if the input array is a valid condensed distance matrix.
`num_obs_dm`(d)	Returns the number of original observations that correspond to a square, redundant distance matrix.
`num_obs_y`(Y)	Returns the number of original observations that correspond to a condensed distance matrix.

`scipy.spatial.distance.`**`is_valid_dm`**(*D*, *tol=0.0*, *throw=False*, *name='D'*, *warning=False*)

Returns True if input array is a valid distance matrix.

Distance matrices must be 2-dimensional numpy arrays. They must have a zero-diagonal, and they must be symmetric.

Parameters **D** : ndarray

The candidate object to test for validity.

tol : float, optional

The distance matrix should be symmetric. *tol* is the maximum difference between entries `ij` and `ji` for the distance metric to be considered symmetric.

throw : bool, optional

An exception is thrown if the distance matrix passed is not valid.

name : str, optional

The name of the variable to checked. This is useful if throw is set to True so the offending variable can be identified in the exception message when an exception is thrown.

warning : bool, optional

Instead of throwing an exception, a warning message is raised.

Returns **valid** : bool

True if the variable *D* passed is a valid distance matrix.

Notes

Small numerical differences in *D* and *D.T* and non-zeroness of the diagonal are ignored if they are within the tolerance specified by *tol*.

`scipy.spatial.distance.`**`is_valid_y`**(*y*, *warning=False*, *throw=False*, *name=None*)

Returns True if the input array is a valid condensed distance matrix.

Condensed distance matrices must be 1-dimensional numpy arrays. Their length must be a binomial coefficient $\binom{n}{2}$ for some positive integer n.

Parameters **y** : ndarray

The condensed distance matrix.

warning : bool, optional

Invokes a warning if the variable passed is not a valid condensed distance matrix. The warning message explains why the distance matrix is not valid. `name` is used when referencing the offending variable.

throw : bool, optional

Throws an exception if the variable passed is not a valid condensed distance matrix.

name : bool, optional

Used when referencing the offending variable in the warning or exception message.

`scipy.spatial.distance.`**`num_obs_dm`**(*d*)

Returns the number of original observations that correspond to a square, redundant distance matrix.

Parameters **d** : ndarray

The target distance matrix.

Returns **num_obs_dm** : int

The number of observations in the redundant distance matrix.

`scipy.spatial.distance.`**`num_obs_y`**(*Y*)

Returns the number of original observations that correspond to a condensed distance matrix.

Parameters **Y** : ndarray

Condensed distance matrix.

Returns **n** : int

The number of observations in the condensed distance matrix *Y*.

Distance functions between two numeric vectors `u` and `v`. Computing distances over a large collection of vectors is inefficient for these functions. Use `pdist` for this purpose.

braycurtis(u, v)	Computes the Bray-Curtis distance between two 1-D arrays.
canberra(u, v)	Computes the Canberra distance between two 1-D arrays.
chebyshev(u, v)	Computes the Chebyshev distance.
cityblock(u, v)	Computes the City Block (Manhattan) distance.
correlation(u, v)	Computes the correlation distance between two 1-D arrays.
cosine(u, v)	Computes the Cosine distance between 1-D arrays.
euclidean(u, v)	Computes the Euclidean distance between two 1-D arrays.
mahalanobis(u, v, VI)	Computes the Mahalanobis distance between two 1-D arrays.
minkowski(u, v, p)	Computes the Minkowski distance between two 1-D arrays.
seuclidean(u, v, V)	Returns the standardized Euclidean distance between two 1-D arrays.
sqeuclidean(u, v)	Computes the squared Euclidean distance between two 1-D arrays.
wminkowski(u, v, p, w)	Computes the weighted Minkowski distance between two 1-D arrays.

scipy.spatial.distance.**braycurtis**(*u*, *v*)

Computes the Bray-Curtis distance between two 1-D arrays.

Bray-Curtis distance is defined as

$$\sum |u_i - v_i| / \sum |u_i + v_i|$$

The Bray-Curtis distance is in the range [0, 1] if all coordinates are positive, and is undefined if the inputs are of length zero.

Parameters

u : (N,) array_like
: Input array.

v : (N,) array_like
: Input array.

Returns

braycurtis : double
: The Bray-Curtis distance between 1-D arrays *u* and *v*.

scipy.spatial.distance.**canberra**(*u*, *v*)

Computes the Canberra distance between two 1-D arrays.

The Canberra distance is defined as

$$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}.$$

Parameters

u : (N,) array_like
: Input array.

v : (N,) array_like
: Input array.

Returns

canberra : double
: The Canberra distance between vectors *u* and *v*.

Notes

When *u[i]* and *v[i]* are 0 for given i, then the fraction 0/0 = 0 is used in the calculation.

scipy.spatial.distance.**chebyshev**(*u*, *v*)

Computes the Chebyshev distance.

Computes the Chebyshev distance between two 1-D arrays *u* and *v*, which is defined as

$$\max_i |u_i - v_i|.$$

Parameters **u** : (N,) array_like

Input vector.

v : (N,) array_like

Input vector.

Returns **chebyshev** : double

The Chebyshev distance between vectors *u* and *v*.

scipy.spatial.distance.**cityblock**(*u*, *v*)

Computes the City Block (Manhattan) distance.

Computes the Manhattan distance between two 1-D arrays *u* and *v*, which is defined as

$$\sum_i |u_i - v_i|.$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **cityblock** : double

The City Block (Manhattan) distance between vectors *u* and *v*.

scipy.spatial.distance.**correlation**(*u*, *v*)

Computes the correlation distance between two 1-D arrays.

The correlation distance between *u* and *v*, is defined as

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{{||(u - \bar{u})||}_2 {||(v - \bar{v})||}_2}$$

where $\bar{u}$ is the mean of the elements of *u* and $x \cdot y$ is the dot product of x and y.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **correlation** : double

The correlation distance between 1-D array *u* and *v*.

scipy.spatial.distance.**cosine**(*u*, *v*)

Computes the Cosine distance between 1-D arrays.

The Cosine distance between *u* and *v*, is defined as

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2}.$$

where $u \cdot v$ is the dot product of u and v.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **cosine** : double

The Cosine distance between vectors *u* and *v*.

scipy.spatial.distance.**euclidean**(*u*, *v*)

Computes the Euclidean distance between two 1-D arrays.

The Euclidean distance between 1-D arrays *u* and *v*, is defined as

$$||u - v||_2$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

Returns **euclidean** : double

The Euclidean distance between vectors *u* and *v*.

scipy.spatial.distance.**mahalanobis**(*u*, *v*, *VI*)

Computes the Mahalanobis distance between two 1-D arrays.

The Mahalanobis distance between 1-D arrays *u* and *v*, is defined as

$$\sqrt{(u - v)V^{-1}(u - v)^T}$$

where `V` is the covariance matrix. Note that the argument *VI* is the inverse of `V`.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

VI : ndarray

The inverse of the covariance matrix.

Returns **mahalanobis** : double

The Mahalanobis distance between vectors *u* and *v*.

scipy.spatial.distance.**minkowski**(*u*, *v*, *p*)

Computes the Minkowski distance between two 1-D arrays.

The Minkowski distance between 1-D arrays *u* and *v*, is defined as

$$||u - v||_p = (\sum |u_i - v_i|^p)^{1/p}.$$

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

p : int

The order of the norm of the difference $||u - v||_p$.

Returns **d** : double

The Minkowski distance between vectors *u* and *v*.

scipy.spatial.distance.**seuclidean**(*u*, *v*, *V*)

Returns the standardized Euclidean distance between two 1-D arrays.

The standardized Euclidean distance between *u* and *v*.

Parameters **u** : (N,) array_like

Input array.

v : (N,) array_like

Input array.

V : (N,) array_like

V is an 1-D array of component variances. It is usually computed among a larger collection vectors.

Returns **seuclidean** : double

The standardized Euclidean distance between vectors *u* and *v*.

`scipy.spatial.distance.sqeuclidean(u, v)`

Computes the squared Euclidean distance between two 1-D arrays.

The squared Euclidean distance between *u* and *v* is defined as

$$||u - v||_2^2.$$

Parameters
- **u** : (N,) array_like
  Input array.
- **v** : (N,) array_like
  Input array.

Returns
- **sqeuclidean** : double
  The squared Euclidean distance between vectors *u* and *v*.

`scipy.spatial.distance.wminkowski(u, v, p, w)`

Computes the weighted Minkowski distance between two 1-D arrays.

The weighted Minkowski distance between *u* and *v*, defined as

$$\left(\sum (|w_i (u_i - v_i)|^p)\right)^{1/p}.$$

Parameters
- **u** : (N,) array_like
  Input array.
- **v** : (N,) array_like
  Input array.
- **p** : int
  The order of the norm of the difference $||u - v||_p$.
- **w** : (N,) array_like
  The weight vector.

Returns
- **wminkowski** : double
  The weighted Minkowski distance between vectors *u* and *v*.

Distance functions between two boolean vectors (representing sets) `u` and `v`. As in the case of numerical vectors, `pdist` is more efficient for computing the distances between all pairs.

dice(u, v)	Computes the Dice dissimilarity between two boolean 1-D arrays.
hamming(u, v)	Computes the Hamming distance between two 1-D arrays.
jaccard(u, v)	Computes the Jaccard-Needham dissimilarity between two boolean 1-D arrays.
kulsinski(u, v)	Computes the Kulsinski dissimilarity between two boolean 1-D arrays.
matching(u, v)	Computes the Hamming distance between two boolean 1-D arrays.
rogerstanimoto(u, v)	Computes the Rogers-Tanimoto dissimilarity between two boolean 1-D arrays.
russellrao(u, v)	Computes the Russell-Rao dissimilarity between two boolean 1-D arrays.
sokalmichener(u, v)	Computes the Sokal-Michener dissimilarity between two boolean 1-D arrays.
sokalsneath(u, v)	Computes the Sokal-Sneath dissimilarity between two boolean 1-D arrays.
yule(u, v)	Computes the Yule dissimilarity between two boolean 1-D arrays.

`scipy.spatial.distance.`**`dice`**(*u*, *v*)

Computes the Dice dissimilarity between two boolean 1-D arrays.

The Dice dissimilarity between *u* and *v*, is

$$\frac{c_{TF} + c_{FT}}{2c_{TT} + c_{FT} + c_{TF}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters **u** : (N,) ndarray, bool
Input 1-D array.
v : (N,) ndarray, bool
Input 1-D array.

Returns **dice** : double
The Dice dissimilarity between 1-D arrays *u* and *v*.

`scipy.spatial.distance.`**`hamming`**(*u*, *v*)

Computes the Hamming distance between two 1-D arrays.

The Hamming distance between 1-D arrays *u* and *v*, is simply the proportion of disagreeing components in *u* and *v*. If *u* and *v* are boolean vectors, the Hamming distance is

$$\frac{c_{01} + c_{10}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters **u** : (N,) array_like
Input array.
v : (N,) array_like
Input array.

Returns **hamming** : double
The Hamming distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`jaccard`**(*u*, *v*)

Computes the Jaccard-Needham dissimilarity between two boolean 1-D arrays.

The Jaccard-Needham dissimilarity between 1-D boolean arrays *u* and *v*, is defined as

$$\frac{c_{TF} + c_{FT}}{c_{TT} + c_{FT} + c_{TF}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters **u** : (N,) array_like, bool
Input array.
v : (N,) array_like, bool
Input array.

Returns **jaccard** : double
The Jaccard distance between vectors *u* and *v*.

`scipy.spatial.distance.`**`kulsinski`**(*u*, *v*)

Computes the Kulsinski dissimilarity between two boolean 1-D arrays.

The Kulsinski dissimilarity between two boolean 1-D arrays *u* and *v*, is defined as

$$\frac{c_{TF} + c_{FT} - c_{TT} + n}{c_{FT} + c_{TF} + n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters **u** : (N,) array_like, bool
Input array.

v : (N,) array_like, bool
Input array.

Returns **kulsinski** : double
The Kulsinski distance between vectors *u* and *v*.

`scipy.spatial.distance.matching(u, v)`

Computes the Hamming distance between two boolean 1-D arrays.

This is a deprecated synonym for *hamming*.

`scipy.spatial.distance.rogerstanimoto(u, v)`

Computes the Rogers-Tanimoto dissimilarity between two boolean 1-D arrays.

The Rogers-Tanimoto dissimilarity between two boolean 1-D arrays *u* and *v*, is defined as

$$\frac{R}{c_{TT} + c_{FF} + R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters **u** : (N,) array_like, bool
Input array.
v : (N,) array_like, bool
Input array.

Returns **rogerstanimoto** : double
The Rogers-Tanimoto dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.russellrao(u, v)`

Computes the Russell-Rao dissimilarity between two boolean 1-D arrays.

The Russell-Rao dissimilarity between two boolean 1-D arrays, *u* and *v*, is defined as

$$\frac{n - c_{TT}}{n}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$.

Parameters **u** : (N,) array_like, bool
Input array.
v : (N,) array_like, bool
Input array.

Returns **russellrao** : double
The Russell-Rao dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.sokalmichener(u, v)`

Computes the Sokal-Michener dissimilarity between two boolean 1-D arrays.

The Sokal-Michener dissimilarity between boolean 1-D arrays *u* and *v*, is defined as

$$\frac{R}{S + R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$, $R = 2 * (c_{TF} + c_{FT})$ and $S = c_{FF} + c_{TT}$.

Parameters **u** : (N,) array_like, bool
Input array.
v : (N,) array_like, bool
Input array.

Returns **sokalmichener** : double
The Sokal-Michener dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`sokalsneath`**`(u, v)`

Computes the Sokal-Sneath dissimilarity between two boolean 1-D arrays.

The Sokal-Sneath dissimilarity between *u* and *v*,

$$\frac{R}{c_{TT} + R}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$.

Parameters **u** : (N,) array_like, bool

Input array.

v : (N,) array_like, bool

Input array.

Returns **sokalsneath** : double

The Sokal-Sneath dissimilarity between vectors *u* and *v*.

`scipy.spatial.distance.`**`yule`**`(u, v)`

Computes the Yule dissimilarity between two boolean 1-D arrays.

The Yule dissimilarity is defined as

$$\frac{R}{c_{TT} * c_{FF} + \frac{R}{2}}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2.0 * c_{TF} * c_{FT}$.

Parameters **u** : (N,) array_like, bool

Input array.

v : (N,) array_like, bool

Input array.

Returns **yule** : double

The Yule dissimilarity between vectors *u* and *v*.

`hamming` also operates over discrete numerical vectors.

5.26 Special functions (`scipy.special`)

Nearly all of the functions below are universal functions and follow broadcasting and automatic array-looping rules. Exceptions are noted.

See also:

`scipy.special.cython_special` – Typed Cython versions of special functions

5.26.1 Error handling

Errors are handled by returning NaNs or other appropriate values. Some of the special function routines can emit warnings or raise exceptions when an error occurs. By default this is disabled; to query and control the current error handling state the following functions are provided.

`geterr`	Get the current way of handling special-function errors.
`seterr`	Set how special-function errors are handled.
`errstate`	Context manager for special-function error handling.
`SpecialFunctionWarning`	Warning that can be emitted by special functions.
`SpecialFunctionError`	Exception that can be raised by special functions.

`scipy.special.geterr()`

Get the current way of handling special-function errors.

Returns **err** : dict

A dictionary with keys “singular”, “underflow”, “overflow”, “slow”, “loss”, “no_result”, “domain”, “arg”, and “other”, whose values are from the strings “ignore”, “warn”, and “raise”. The keys represent possible special-function errors, and the values define how these errors are handled.

See also:

seterr set how special-function errors are handled

errstate context manager for special-function error handling

numpy.geterr similar numpy function for floating-point errors

Notes

For complete documentation of the types of special-function errors and treatment options, see *seterr*.

Examples

By default all errors are ignored.

```
>>> import scipy.special as sc
>>> for key, value in sorted(sc.geterr().items()):
...     print("{}: {}".format(key, value))
...
arg: ignore
domain: ignore
loss: ignore
no_result: ignore
other: ignore
overflow: ignore
singular: ignore
slow: ignore
underflow: ignore
```

`scipy.special.seterr()`

Set how special-function errors are handled.

Parameters **all** : {‘ignore’, ‘warn’ ‘raise’}, optional

Set treatment for all type of special-function errors at once. The options are:

- ‘ignore’ Take no action when the error occurs
- ‘warn’ Print a *SpecialFunctionWarning* when the error occurs (via the Python `warnings` module)
- ‘raise’ Raise a *SpecialFunctionError* when the error occurs.

The default is to not change the current behavior. If behaviors for additional categories of special-function errors are specified, then `all` is applied first, followed by the additional categories.

singular : {‘ignore’, ‘warn’, ‘raise’}, optional

Treatment for singularities.

underflow : {‘ignore’, ‘warn’, ‘raise’}, optional

Treatment for underflow.

overflow : {‘ignore’, ‘warn’, ‘raise’}, optional

Treatment for overflow.

slow : {‘ignore’, ‘warn’, ‘raise’}, optional

Treatment for slow convergence.

loss : {'ignore', 'warn', 'raise'}, optional
Treatment for loss of accuracy.

no_result : {'ignore', 'warn', 'raise'}, optional
Treatment for failing to find a result.

domain : {'ignore', 'warn', 'raise'}, optional
Treatment for an invalid argument to a function.

arg : {'ignore', 'warn', 'raise'}, optional
Treatment for an invalid parameter to a function.

other : {'ignore', 'warn', 'raise'}, optional
Treatment for an unknown error.

Returns **olderr** : dict
Dictionary containing the old settings.

See also:

geterr get the current way of handling special-function errors

errstate context manager for special-function error handling

numpy.seterr
similar numpy function for floating-point errors

Examples

```
>>> import scipy.special as sc
>>> from numpy.testing import assert_raises
>>> sc.gammaln(0)
inf
>>> olderr = sc.seterr(singular='raise')
>>> with assert_raises(sc.SpecialFunctionError):
...     sc.gammaln(0)
...
>>> _ = sc.seterr(**olderr)
```

We can also raise for every category except one.

```
>>> olderr = sc.seterr(all='raise', singular='ignore')
>>> sc.gammaln(0)
inf
>>> with assert_raises(sc.SpecialFunctionError):
...     sc.spence(-1)
...
>>> _ = sc.seterr(**olderr)
```

class scipy.special.**errstate**
Context manager for special-function error handling.

Using an instance of *errstate* as a context manager allows statements in that context to execute with a known error handling behavior. Upon entering the context the error handling is set with *seterr*, and upon exiting it is restored to what it was before.

Parameters **kwargs** : {all, singular, underflow, overflow, slow, loss, no_result, domain, arg, other}
Keyword arguments. The valid keywords are possible special-function errors. Each keyword should have a string value that defines the treatement for the particular type of error. Values must be 'ignore', 'warn', or 'other'. See *seterr* for details.

See also:

geterr get the current way of handling special-function errors

seterr set how special-function errors are handled

numpy.errstate
similar numpy function for floating-point errors

Examples

```
>>> import scipy.special as sc
>>> from numpy.testing import assert_raises
>>> sc.gammaln(0)
inf
>>> with sc.errstate(singular='raise'):
...     with assert_raises(sc.SpecialFunctionError):
...         sc.gammaln(0)
...
>>> sc.gammaln(0)
inf
```

We can also raise on every category except one.

```
>>> with sc.errstate(all='raise', singular='ignore'):
...     sc.gammaln(0)
...     with assert_raises(sc.SpecialFunctionError):
...         sc.spence(-1)
...
inf
```

exception scipy.special.**SpecialFunctionWarning**
Warning that can be emitted by special functions.

exception scipy.special.**SpecialFunctionError**
Exception that can be raised by special functions.

5.26.2 Available functions

Airy functions

airy(z)	Airy functions and their derivatives.
airye(z)	Exponentially scaled Airy functions and their derivatives.
ai_zeros(nt)	Compute *nt* zeros and values of the Airy function Ai and its derivative.
bi_zeros(nt)	Compute *nt* zeros and values of the Airy function Bi and its derivative.
itairy(x)	Integrals of Airy functions

scipy.special.**airy**(*z*) = <ufunc 'airy'>
Airy functions and their derivatives.

Parameters **z** : array_like
Real or complex argument.

Returns **Ai, Aip, Bi, Bip** : ndarrays
Airy functions Ai and Bi, and their derivatives Aip and Bip.

See also:

airye exponentially scaled Airy functions.

Notes

The Airy functions Ai and Bi are two independent solutions of

$$y''(x) = xy(x).$$

For real z in [-10, 10], the computation is carried out by calling the Cephes *[R368]* `airy` routine, which uses power series summation for small z and rational minimax approximations for large z.

Outside this range, the AMOS *[R369]* *zairy* and *zbiry* routines are employed. They are computed using power series for $|z| < 1$ and the following relations to modified Bessel functions for larger z (where $t \equiv 2z^{3/2}/3$):

$$Ai(z) = \frac{1}{\pi\sqrt{3}} K_{1/3}(t)$$
$$Ai'(z) = -\frac{z}{\pi\sqrt{3}} K_{2/3}(t)$$
$$Bi(z) = \sqrt{\frac{z}{3}} \left(I_{-1/3}(t) + I_{1/3}(t)\right)$$
$$Bi'(z) = \frac{z}{\sqrt{3}} \left(I_{-2/3}(t) + I_{2/3}(t)\right)$$

References

[R368], *[R369]*

`scipy.special.`**`airye`**`(z)` = **<ufunc 'airye'>**

Exponentially scaled Airy functions and their derivatives.

Scaling:

```
eAi  = Ai  * exp(2.0/3.0*z*sqrt(z))
eAip = Aip * exp(2.0/3.0*z*sqrt(z))
eBi  = Bi  * exp(-abs(2.0/3.0*(z*sqrt(z)).real))
eBip = Bip * exp(-abs(2.0/3.0*(z*sqrt(z)).real))
```

Parameters **z** : array_like

Real or complex argument.

Returns **eAi, eAip, eBi, eBip** : array_like

Airy functions Ai and Bi, and their derivatives Aip and Bip

See also:

`airy`

Notes

Wrapper for the AMOS *[R370]* routines *zairy* and *zbiry*.

References

[R370]

`scipy.special.`**`ai_zeros`**`(nt)`

Compute *nt* zeros and values of the Airy function Ai and its derivative.

Computes the first *nt* zeros, *a*, of the Airy function Ai(x); first *nt* zeros, *ap*, of the derivative of the Airy function Ai'(x); the corresponding values Ai(a'); and the corresponding values Ai'(a).

Parameters **nt** : int

Number of zeros to compute

Returns **a** : ndarray

First *nt* zeros of Ai(x)

ap : ndarray

First *nt* zeros of Ai'(x)

ai : ndarray

Values of Ai(x) evaluated at first *nt* zeros of Ai'(x)

aip : ndarray

Values of Ai'(x) evaluated at first *nt* zeros of Ai(x)

References

[R367]

`scipy.special.bi_zeros(nt)`

Compute *nt* zeros and values of the Airy function Bi and its derivative.

Computes the first *nt* zeros, b, of the Airy function Bi(x); first *nt* zeros, b', of the derivative of the Airy function Bi'(x); the corresponding values Bi(b'); and the corresponding values Bi'(b).

Parameters **nt** : int

Number of zeros to compute

Returns **b** : ndarray

First *nt* zeros of Bi(x)

bp : ndarray

First *nt* zeros of Bi'(x)

bi : ndarray

Values of Bi(x) evaluated at first *nt* zeros of Bi'(x)

bip : ndarray

Values of Bi'(x) evaluated at first *nt* zeros of Bi(x)

References

[R383]

`scipy.special.itairy(x)` = <ufunc 'itairy'>

Integrals of Airy functions

Calculates the integrals of Airy functions from 0 to *x*.

Parameters **x: array_like**

Upper limit of integration (float).

Returns Apt

Integral of Ai(t) from 0 to x.

Bpt

Integral of Bi(t) from 0 to x.

Ant

Integral of Ai(-t) from 0 to x.

Bnt

Integral of Bi(-t) from 0 to x.

Notes

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R444]*.

References

[R444]

Elliptic Functions and Integrals

ellipj(u, m)	Jacobian elliptic functions
ellipk(m)	Complete elliptic integral of the first kind.
ellipkm1(p)	Complete elliptic integral of the first kind around *m* = 1
ellipkinc(phi, m)	Incomplete elliptic integral of the first kind
ellipe(m)	Complete elliptic integral of the second kind
ellipeinc(phi, m)	Incomplete elliptic integral of the second kind

scipy.special.**ellipj**(*u*, *m*) = <ufunc 'ellipj'>

Jacobian elliptic functions

Calculates the Jacobian elliptic functions of parameter *m* between 0 and 1, and real argument *u*.

Parameters

m : array_like
: Parameter.

u : array_like
: Argument.

Returns

sn, cn, dn, ph : ndarrays
: The returned functions:

```
sn(u|m), cn(u|m), dn(u|m)
```

The value *ph* is such that if *u = ellipk(ph, m)*, then *sn(u|m) = sin(ph)* and *cn(u|m) = cos(ph)*.

See also:

ellipk Complete elliptic integral of the first kind.

Notes

Wrapper for the Cephes *[R404]* routine *ellpj*.

These functions are periodic, with quarter-period on the real axis equal to the complete elliptic integral *ellipk(m)*.

Relation to incomplete elliptic integral: If *u = ellipk(phi,m)*, then *sn(u|m) = sin(phi)*, and *cn(u|m) = cos(phi)*. The *phi* is called the amplitude of *u*.

Computation is by means of the arithmetic-geometric mean algorithm, except when *m* is within 1e-9 of 0 or 1. In the latter case with *m* close to 1, the approximation applies only for *phi < pi/2*.

References

[R404]

scipy.special.**ellipk**(*m*)

Complete elliptic integral of the first kind.

This function is defined as

$$K(m) = \int_0^{\pi/2} [1 - m \sin(t)^2]^{-1/2} dt$$

Parameters

m : array_like
: The parameter of the elliptic integral.

Returns

K : array_like
: Value of the elliptic integral.

See also:

ellipkm1 Complete elliptic integral of the first kind around m = 1
ellipkinc Incomplete elliptic integral of the first kind

ellipe	Complete elliptic integral of the second kind
ellipeinc	Incomplete elliptic integral of the second kind

Notes

For more precision around point m = 1, use *ellipkm1*, which this function calls.

`scipy.special.ellipkm1`(*p*) = <ufunc 'ellipkm1'>

Complete elliptic integral of the first kind around *m* = 1

This function is defined as

$$K(p) = \int_0^{\pi/2} [1 - m \sin(t)^2]^{-1/2} dt$$

where *m = 1 - p*.

Parameters **p** : array_like

Defines the parameter of the elliptic integral as *m = 1 - p*.

Returns **K** : ndarray

Value of the elliptic integral.

See also:

ellipk	Complete elliptic integral of the first kind
ellipkinc	Incomplete elliptic integral of the first kind
ellipe	Complete elliptic integral of the second kind
ellipeinc	Incomplete elliptic integral of the second kind

Notes

Wrapper for the Cephes *[R406]* routine *ellpk*.

For *p <= 1*, computation uses the approximation,

$$K(p) \approx P(p) - \log(p) Q(p),$$

where P and Q are tenth-order polynomials. The argument *p* is used internally rather than *m* so that the logarithmic singularity at *m = 1* will be shifted to the origin; this preserves maximum accuracy. For *p > 1*, the identity

$$K(p) = K(1/p)/\sqrt(p)$$

is used.

References

[R406]

`scipy.special.ellipkinc`(*phi*, *m*) = <ufunc 'ellipkinc'>

Incomplete elliptic integral of the first kind

This function is defined as

$$K(\phi, m) = \int_0^{\phi} [1 - m \sin(t)^2]^{-1/2} dt$$

This function is also called *F(phi, m)*.

Parameters **phi** : array_like

amplitude of the elliptic integral

m : array_like

Returns **K** : ndarray

parameter of the elliptic integral

Value of the elliptic integral

See also:

ellipkm1 Complete elliptic integral of the first kind, near *m* = 1
ellipk Complete elliptic integral of the first kind
ellipe Complete elliptic integral of the second kind
ellipeinc Incomplete elliptic integral of the second kind

Notes

Wrapper for the Cephes *[R405]* routine *ellik*. The computation is carried out using the arithmetic-geometric mean algorithm.

References

[R405]

`scipy.special.`**`ellipe`**(*m*) **= <ufunc 'ellipe'>**

Complete elliptic integral of the second kind

This function is defined as

$$E(m) = \int_0^{\pi/2} [1 - m \sin(t)^2]^{1/2} dt$$

Parameters **m** : array_like

Defines the parameter of the elliptic integral.

Returns **E** : ndarray

Value of the elliptic integral.

See also:

ellipkm1 Complete elliptic integral of the first kind, near *m* = 1
ellipk Complete elliptic integral of the first kind
ellipkinc Incomplete elliptic integral of the first kind
ellipeinc Incomplete elliptic integral of the second kind

Notes

Wrapper for the Cephes *[R402]* routine *ellpe*.

For *m > 0* the computation uses the approximation,

$$E(m) \approx P(1-m) - (1-m)\log(1-m)Q(1-m),$$

where P and Q are tenth-order polynomials. For *m < 0*, the relation

$$E(m) = E(m/(m-1))\sqrt(1-m)$$

is used.

References

[R402]

`scipy.special.`**`ellipeinc`**(*phi*, *m*) **= <ufunc 'ellipeinc'>**

Incomplete elliptic integral of the second kind

This function is defined as

$$E(\phi, m) = \int_0^{\phi} [1 - m \sin(t)^2]^{1/2} dt$$

Parameters **phi** : array_like

amplitude of the elliptic integral.

m : array_like

parameter of the elliptic integral.

Returns **E** : ndarray

Value of the elliptic integral.

See also:

ellipkm1	Complete elliptic integral of the first kind, near $m = 1$
ellipk	Complete elliptic integral of the first kind
ellipkinc	Incomplete elliptic integral of the first kind
ellipe	Complete elliptic integral of the second kind

Notes

Wrapper for the Cephes *[R403]* routine *ellie*.

Computation uses arithmetic-geometric means algorithm.

References

[R403]

Bessel Functions

jv(v, z)	Bessel function of the first kind of real order and complex argument.
jn(v, z)	Bessel function of the first kind of real order and complex argument.
jve(v, z)	Exponentially scaled Bessel function of order *v*.
yn(n, x)	Bessel function of the second kind of integer order and real argument.
yv(v, z)	Bessel function of the second kind of real order and complex argument.
yve(v, z)	Exponentially scaled Bessel function of the second kind of real order.
kn(n, x)	Modified Bessel function of the second kind of integer order *n*
kv(v, z)	Modified Bessel function of the second kind of real order *v*
kve(v, z)	Exponentially scaled modified Bessel function of the second kind.
iv(v, z)	Modified Bessel function of the first kind of real order.
ive(v, z)	Exponentially scaled modified Bessel function of the first kind
hankel1(v, z)	Hankel function of the first kind
hankel1e(v, z)	Exponentially scaled Hankel function of the first kind
hankel2(v, z)	Hankel function of the second kind
hankel2e(v, z)	Exponentially scaled Hankel function of the second kind

scipy.special.**jv**(*v*, *z*) = **<ufunc 'jv'>**

Bessel function of the first kind of real order and complex argument.

Parameters **v** : array_like

Order (float).

	z : array_like	Argument (float or complex).
Returns	**J** : ndarray	Value of the Bessel function, $J_v(z)$.

See also:

jve J_v with leading exponential behavior stripped off.

spherical_jn spherical Bessel functions.

Notes

For positive *v* values, the computation is carried out using the AMOS *[R459]* *zbesj* routine, which exploits the connection to the modified Bessel function I_v,

$$J_v(z) = \exp(n\pi\imath/2) I_v(-\imath z) \qquad (\Im z > 0)$$
$$J_v(z) = \exp(-n\pi\imath/2) I_v(\imath z) \qquad (\Im z < 0)$$

For negative *v* values the formula,

$$J_{-v}(z) = J_v(z)\cos(\pi v) - Y_v(z)\sin(\pi v)$$

is used, where $Y_v(z)$ is the Bessel function of the second kind, computed using the AMOS routine *zbesy*. Note that the second term is exactly zero for integer *v*; to improve accuracy the second term is explicitly omitted for *v* values such that *v = floor(v)*.

Not to be confused with the spherical Bessel functions (see `spherical_jn`).

References

[R459]

`scipy.special.`**`jn`**(*v*, *z*) = **<ufunc 'jv'>**

Bessel function of the first kind of real order and complex argument.

Parameters	**v** : array_like	Order (float).
	z : array_like	Argument (float or complex).
Returns	**J** : ndarray	Value of the Bessel function, $J_v(z)$.

See also:

jve J_v with leading exponential behavior stripped off.

spherical_jn spherical Bessel functions.

Notes

For positive *v* values, the computation is carried out using the AMOS *[R454]* *zbesj* routine, which exploits the connection to the modified Bessel function I_v,

$$J_v(z) = \exp(n\pi\imath/2) I_v(-\imath z) \qquad (\Im z > 0)$$
$$J_v(z) = \exp(-n\pi\imath/2) I_v(\imath z) \qquad (\Im z < 0)$$

For negative *v* values the formula,

$$J_{-v}(z) = J_v(z)\cos(\pi v) - Y_v(z)\sin(\pi v)$$

is used, where $Y_v(z)$ is the Bessel function of the second kind, computed using the AMOS routine *zbesy*. Note that the second term is exactly zero for integer *v*; to improve accuracy the second term is explicitly omitted for *v* values such that *v = floor(v)*.

Not to be confused with the spherical Bessel functions (see `spherical_jn`).

References

[R454]

`scipy.special.`**`jve`**(*v*, *z*) **= <ufunc 'jve'>**

Exponentially scaled Bessel function of order *v*.

Defined as:

```
jve(v, z) = jv(v, z) * exp(-abs(z.imag))
```

Parameters **v** : array_like

Order (float).

z : array_like

Argument (float or complex).

Returns **J** : ndarray

Value of the exponentially scaled Bessel function.

Notes

For positive *v* values, the computation is carried out using the AMOS *[R460] zbesj* routine, which exploits the connection to the modified Bessel function I_v,

$$J_v(z) = \exp(n\pi\imath/2)I_v(-\imath z) \qquad (\Im z > 0)$$
$$J_v(z) = \exp(-n\pi\imath/2)I_v(\imath z) \qquad (\Im z < 0)$$

For negative *v* values the formula,

$$J_{-v}(z) = J_v(z)\cos(\pi v) - Y_v(z)\sin(\pi v)$$

is used, where $Y_v(z)$ is the Bessel function of the second kind, computed using the AMOS routine *zbesy*. Note that the second term is exactly zero for integer *v*; to improve accuracy the second term is explicitly omitted for *v* values such that *v = floor(v)*.

References

[R460]

`scipy.special.`**`yn`**(*n*, *x*) **= <ufunc 'yn'>**

Bessel function of the second kind of integer order and real argument.

Parameters **n** : array_like

Order (integer).

z : array_like

Argument (float).

Returns **Y** : ndarray

Value of the Bessel function, $Y_n(x)$.

See also:

`yv` For real order and real or complex argument.

Notes

Wrapper for the Cephes *[R540]* routine `yn`.

The function is evaluated by forward recurrence on *n*, starting with values computed by the Cephes routines `y0` and `y1`. If *n = 0* or 1, the routine for `y0` or `y1` is called directly.

References

[R540]

`scipy.special.`**`yv`**(*v*, *z*) **= <ufunc 'yv'>**

Bessel function of the second kind of real order and complex argument.

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **Y** : ndarray
  Value of the Bessel function of the second kind, $Y_v(x)$.

See also:

yve Y_v with leading exponential behavior stripped off.

Notes

For positive *v* values, the computation is carried out using the AMOS *[R543]* *zbesy* routine, which exploits the connection to the Hankel Bessel functions $H_v^{(1)}$ and $H_v^{(2)}$,

$$Y_v(z) = \frac{1}{2\imath}(H_v^{(1)} - H_v^{(2)}).$$

For negative *v* values the formula,

$$Y_{-v}(z) = Y_v(z)\cos(\pi v) + J_v(z)\sin(\pi v)$$

is used, where $J_v(z)$ is the Bessel function of the first kind, computed using the AMOS routine *zbesj*. Note that the second term is exactly zero for integer *v*; to improve accuracy the second term is explicitly omitted for *v* values such that *v = floor(v)*.

References

[R543]

`scipy.special.`**`yve`**(*v*, *z*) **= <ufunc 'yve'>**

Exponentially scaled Bessel function of the second kind of real order.

Returns the exponentially scaled Bessel function of the second kind of real order *v* at complex *z*:

```
yve(v, z) = yv(v, z) * exp(-abs(z.imag))
```

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **Y** : ndarray
  Value of the exponentially scaled Bessel function.

Notes

For positive *v* values, the computation is carried out using the AMOS *[R544]* *zbesy* routine, which exploits the connection to the Hankel Bessel functions $H_v^{(1)}$ and $H_v^{(2)}$,

$$Y_v(z) = \frac{1}{2\imath}(H_v^{(1)} - H_v^{(2)}).$$

For negative *v* values the formula,

$$Y_{-v}(z) = Y_v(z)\cos(\pi v) + J_v(z)\sin(\pi v)$$

is used, where $J_v(z)$ is the Bessel function of the first kind, computed using the AMOS routine *zbesj*. Note that the second term is exactly zero for integer *v*; to improve accuracy the second term is explicitly omitted for *v* values such that *v = floor(v)*.

References

[R544]

`scipy.special.`**`kn`**(*n*, *x*) **= <ufunc 'kn'>**

Modified Bessel function of the second kind of integer order *n*

Returns the modified Bessel function of the second kind for integer order *n* at real *z*.

These are also sometimes called functions of the third kind, Basset functions, or Macdonald functions.

Parameters

n : array_like of int
: Order of Bessel functions (floats will truncate with a warning)

z : array_like of float
: Argument at which to evaluate the Bessel functions

Returns

out : ndarray
: The results

See also:

kv	Same function, but accepts real order and complex argument
kvp	Derivative of this function

Notes

Wrapper for AMOS *[R471]* routine *zbesk*. For a discussion of the algorithm used, see *[R472]* and the references therein.

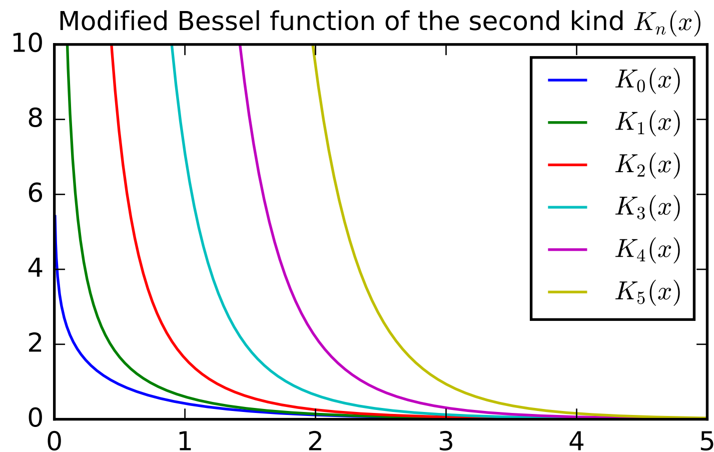
References

[R471], *[R472]*

Examples

Plot the function of several orders for real input:

```
>>> from scipy.special import kn
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 5, 1000)
>>> for N in range(6):
...     plt.plot(x, kn(N, x), label='$K_{}(x)$'.format(N))
>>> plt.ylim(0, 10)
>>> plt.legend()
>>> plt.title(r'Modified Bessel function of the second kind $K_n(x)$')
>>> plt.show()
```



Calculate for a single value at multiple orders:

```
>>> kn([4, 5, 6], 1)
array([   44.23241585,    360.9605896 ,   3653.83831186])
```

scipy.special.**kv**(*v*, *z*) = <ufunc 'kv'>

Modified Bessel function of the second kind of real order *v*

Returns the modified Bessel function of the second kind for real order *v* at complex *z*.

These are also sometimes called functions of the third kind, Basset functions, or Macdonald functions. They are defined as those solutions of the modified Bessel equation for which,

$$K_v(x) \sim \sqrt{\pi/(2x)} \exp(-x)$$

as $x \to \infty$ *[R475]*.

Parameters

v : array_like of float
Order of Bessel functions

z : array_like of complex
Argument at which to evaluate the Bessel functions

Returns

out : ndarray
The results. Note that input must be of complex type to get complex output, e.g. `kv(3, -2+0j)` instead of `kv(3, -2)`.

See also:

kve This function with leading exponential behavior stripped off.
kvp Derivative of this function

Notes

Wrapper for AMOS *[R473]* routine *zbesk*. For a discussion of the algorithm used, see *[R474]* and the references therein.

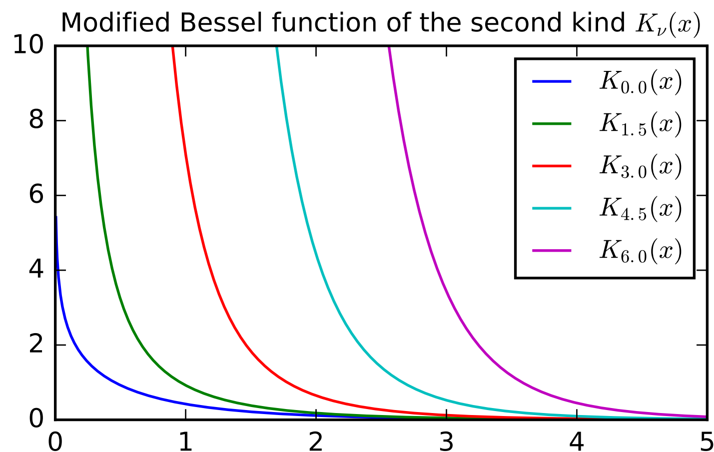
References

[R473], *[R474]*, *[R475]*

Examples

Plot the function of several orders for real input:

```
>>> from scipy.special import kv
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 5, 1000)
>>> for N in np.linspace(0, 6, 5):
...     plt.plot(x, kv(N, x), label='$K_{{{}}}(x)$'.format(N))
>>> plt.ylim(0, 10)
>>> plt.legend()
>>> plt.title(r'Modified Bessel function of the second kind $K_\nu(x)$')
>>> plt.show()
```



Calculate for a single value at multiple orders:

```
>>> kv([4, 4.5, 5], 1+2j)
array([ 0.1992+2.3892j,  2.3493+3.6j   ,  7.2827+3.8104j])
```

`scipy.special.kve(v, z) = <ufunc 'kve'>`

Exponentially scaled modified Bessel function of the second kind.

Returns the exponentially scaled, modified Bessel function of the second kind (sometimes called the third kind) for real order *v* at complex *z*:

```
kve(v, z) = kv(v, z) * exp(z)
```

Parameters
- **v** : array_like of float
  Order of Bessel functions
- **z** : array_like of complex
  Argument at which to evaluate the Bessel functions

Returns
- **out** : ndarray
  The exponentially scaled modified Bessel function of the second kind.

Notes

Wrapper for AMOS *[R476]* routine *zbesk*. For a discussion of the algorithm used, see *[R477]* and the references therein.

References

[R476], *[R477]*

`scipy.special.iv(v, z)` **= <ufunc 'iv'>**

Modified Bessel function of the first kind of real order.

Parameters

v : array_like
: Order. If z is of real type and negative, v must be integer valued.

z : array_like of float or complex
: Argument.

Returns

out : ndarray
: Values of the modified Bessel function.

See also:

kve This function with leading exponential behavior stripped off.

Notes

For real z and $v \in [-50, 50]$, the evaluation is carried out using Temme's method *[R447]*. For larger orders, uniform asymptotic expansions are applied.

For complex z and positive v, the AMOS *[R448]* *zbesi* routine is called. It uses a power series for small z, the asymptitic expansion for large *abs(z)*, the Miller algorithm normalized by the Wronskian and a Neumann series for intermediate magnitudes, and the uniform asymptitic expansions for $I_v(z)$ and $J_v(z)$ for large orders. Backward recurrence is used to generate sequences or reduce orders when necessary.

The calculations above are done in the right half plane and continued into the left half plane by the formula,

$$I_v(z \exp(\pm \imath \pi)) = \exp(\pm \pi v) I_v(z)$$

(valid when the real part of z is positive). For negative v, the formula

$$I_{-v}(z) = I_v(z) + \frac{2}{\pi} \sin(\pi v) K_v(z)$$

is used, where $K_v(z)$ is the modified Bessel function of the second kind, evaluated using the AMOS routine *zbesk*.

References

[R447], *[R448]*

`scipy.special.ive(v, z)` **= <ufunc 'ive'>**

Exponentially scaled modified Bessel function of the first kind

Defined as:

```
ive(v, z) = iv(v, z) * exp(-abs(z.real))
```

Parameters

v : array_like of float
: Order.

z : array_like of float or complex
: Argument.

Returns

out : ndarray
: Values of the exponentially scaled modified Bessel function.

Notes

For positive *v*, the AMOS *[R449]* *zbesi* routine is called. It uses a power series for small *z*, the asymptitic expansion for large *abs(z)*, the Miller algorithm normalized by the Wronskian and a Neumann series for intermediate magnitudes, and the uniform asymptitic expansions for $I_v(z)$ and $J_v(z)$ for large orders. Backward recurrence is used to generate sequences or reduce orders when necessary.

The calculations above are done in the right half plane and continued into the left half plane by the formula,

$$I_v(z \exp(\pm\imath\pi)) = \exp(\pm\pi v) I_v(z)$$

(valid when the real part of *z* is positive). For negative *v*, the formula

$$I_{-v}(z) = I_v(z) + \frac{2}{\pi} \sin(\pi v) K_v(z)$$

is used, where $K_v(z)$ is the modified Bessel function of the second kind, evaluated using the AMOS routine *zbesk*.

References

[R449]

`scipy.special.hankel1(v, z) = <ufunc 'hankel1'>`

Hankel function of the first kind

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **out** : Values of the Hankel function of the first kind.

See also:

hankel1e this function with leading exponential behavior stripped off.

Notes

A wrapper for the AMOS *[R435]* routine *zbesh*, which carries out the computation using the relation,

$$H_v^{(1)}(z) = \frac{2}{\imath\pi} \exp(-\imath\pi v/2) K_v(z \exp(-\imath\pi/2))$$

where K_v is the modified Bessel function of the second kind. For negative orders, the relation

$$H_{-v}^{(1)}(z) = H_v^{(1)}(z) \exp(\imath\pi v)$$

is used.

References

[R435]

`scipy.special.hankel1e(v, z) = <ufunc 'hankel1e'>`

Exponentially scaled Hankel function of the first kind

Defined as:

```
hankel1e(v, z) = hankel1(v, z) * exp(-1j * z)
```

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **out** : Values of the exponentially scaled Hankel function.

Notes

A wrapper for the AMOS [R436] routine *zbesh*, which carries out the computation using the relation,

$$H^{(1)}_v(z) = \frac{2}{\imath\pi} \exp(-\imath \pi v/2) K_v(z \exp(-\imath\pi/2))$$

where K_v is the modified Bessel function of the second kind. For negative orders, the relation

$$H^{(1)}_{-v}(z) = H^{(1)}_v(z) \exp(\imath\pi v)$$

is used.

References

[R436]

`scipy.special.hankel2(v, z) = <ufunc 'hankel2'>`

Hankel function of the second kind

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **out** : Values of the Hankel function of the second kind.

See also:

hankel2e this function with leading exponential behavior stripped off.

Notes

A wrapper for the AMOS [R437] routine *zbesh*, which carries out the computation using the relation,

$$H^{(2)}_v(z) = -\frac{2}{\imath\pi} \exp(\imath \pi v/2) K_v(z \exp(\imath\pi/2))$$

where K_v is the modified Bessel function of the second kind. For negative orders, the relation

$$H^{(2)}_{-v}(z) = H^{(2)}_v(z) \exp(-\imath\pi v)$$

is used.

References

[R437]

`scipy.special.hankel2e(v, z) = <ufunc 'hankel2e'>`

Exponentially scaled Hankel function of the second kind

Defined as:

```
hankel2e(v, z) = hankel2(v, z) * exp(1j * z)
```

Parameters
- **v** : array_like
  Order (float).
- **z** : array_like
  Argument (float or complex).

Returns
- **out** : Values of the exponentially scaled Hankel function of the second kind.

Notes

A wrapper for the AMOS *[R438]* routine *zbesh*, which carries out the computation using the relation,

$$H^{(2)}_v(z) = -\frac{2}{\imath\pi} \exp(\frac{\imath\pi v}{2}) K_v(z exp(\frac{\imath\pi}{2}))$$

where K_v is the modified Bessel function of the second kind. For negative orders, the relation

$$H^{(2)}_{-v}(z) = H^{(2)}_v(z) \exp(-\imath\pi v)$$

is used.

References

[R438]

The following is not an universal function:

lmbda(v, x)	Jahnke-Emden Lambda function, Lambdav(x).

`scipy.special.lmbda(v, x)`

Jahnke-Emden Lambda function, Lambdav(x).

This function is defined as *[R483]*,

$$\Lambda_v(x) = \Gamma(v+1) \frac{J_v(x)}{(x/2)^v},$$

where Γ is the gamma function and J_v is the Bessel function of the first kind.

Parameters

v : float

Order of the Lambda function

x : float

Value at which to evaluate the function and derivatives

Returns

vl : ndarray

Values of Lambda_vi(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.

dl : ndarray

Derivatives Lambda_vi'(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.

References

[R482], *[R483]*

Zeros of Bessel Functions

These are not universal functions:

jnjnp_zeros(nt)	Compute zeros of integer-order Bessel functions Jn and Jn'.
jnyn_zeros(n, nt)	Compute nt zeros of Bessel functions Jn(x), Jn'(x), Yn(x), and Yn'(x).
jn_zeros(n, nt)	Compute zeros of integer-order Bessel function Jn(x).
jnp_zeros(n, nt)	Compute zeros of integer-order Bessel function derivative Jn'(x).
yn_zeros(n, nt)	Compute zeros of integer-order Bessel function Yn(x).
ynp_zeros(n, nt)	Compute zeros of integer-order Bessel function derivative Yn'(x).

Continued on next page

Table 5.235 – continued from previous page

y0_zeros(nt[, complex])	Compute nt zeros of Bessel function Y0(z), and derivative at each zero.
y1_zeros(nt[, complex])	Compute nt zeros of Bessel function Y1(z), and derivative at each zero.
y1p_zeros(nt[, complex])	Compute nt zeros of Bessel derivative Y1'(z), and value at each zero.

scipy.special.**jnjnp_zeros**(*nt*)

Compute zeros of integer-order Bessel functions Jn and Jn'.

Results are arranged in order of the magnitudes of the zeros.

Parameters **nt** : int

Number (<=1200) of zeros to compute

Returns **zo[l-1]** : ndarray

Value of the lth zero of Jn(x) and Jn'(x). Of length *nt*.

n[l-1] : ndarray

Order of the Jn(x) or Jn'(x) associated with lth zero. Of length *nt*.

m[l-1] : ndarray

Serial number of the zeros of Jn(x) or Jn'(x) associated with lth zero. Of length *nt*.

t[l-1] : ndarray

0 if lth zero in zo is zero of Jn(x), 1 if it is a zero of Jn'(x). Of length *nt*.

See also:

jn_zeros, *jnp_zeros*

References

[R456]

scipy.special.**jnyn_zeros**(*n*, *nt*)

Compute nt zeros of Bessel functions Jn(x), Jn'(x), Yn(x), and Yn'(x).

Returns 4 arrays of length *nt*, corresponding to the first *nt* zeros of Jn(x), Jn'(x), Yn(x), and Yn'(x), respectively.

Parameters **n** : int

Order of the Bessel functions

nt : int

Number (<=1200) of zeros to compute

See jn_zeros, jnp_zeros, yn_zeros, ynp_zeros to get separate arrays.

References

[R458]

scipy.special.**jn_zeros**(*n*, *nt*)

Compute zeros of integer-order Bessel function Jn(x).

Parameters **n** : int

Order of Bessel function

nt : int

Number of zeros to return

References

[R455]

`scipy.special.`**`jnp_zeros`**(*n*, *nt*)

Compute zeros of integer-order Bessel function derivative Jn'(x).

Parameters **n** : int
: Order of Bessel function

nt : int
: Number of zeros to return

References

[R457]

`scipy.special.`**`yn_zeros`**(*n*, *nt*)

Compute zeros of integer-order Bessel function Yn(x).

Parameters **n** : int
: Order of Bessel function

nt : int
: Number of zeros to return

References

[R541]

`scipy.special.`**`ynp_zeros`**(*n*, *nt*)

Compute zeros of integer-order Bessel function derivative Yn'(x).

Parameters **n** : int
: Order of Bessel function

nt : int
: Number of zeros to return

References

[R542]

`scipy.special.`**`y0_zeros`**(*nt*, *complex=False*)

Compute nt zeros of Bessel function Y0(z), and derivative at each zero.

The derivatives are given by Y0'(z0) = -Y1(z0) at each zero z0.

Parameters **nt** : int
: Number of zeros to return

complex : bool, default False
: Set to False to return only the real zeros; set to True to return only the complex zeros with negative real part and positive imaginary part. Note that the complex conjugates of the latter are also zeros of the function, but are not returned by this routine.

Returns **z0n** : ndarray
: Location of nth zero of Y0(z)

y0pz0n : ndarray
: Value of derivative Y0'(z0) for nth zero

References

[R536]

`scipy.special.`**`y1_zeros`**(*nt*, *complex=False*)

Compute nt zeros of Bessel function Y1(z), and derivative at each zero.

The derivatives are given by Y1'(z1) = Y0(z1) at each zero z1.

Parameters **nt** : int

Number of zeros to return

complex : bool, default False

Set to False to return only the real zeros; set to True to return only the complex zeros with negative real part and positive imaginary part. Note that the complex conjugates of the latter are also zeros of the function, but are not returned by this routine.

Returns **z1n** : ndarray

Location of nth zero of Y1(z)

y1pz1n : ndarray

Value of derivative Y1'(z1) for nth zero

References

[R538]

`scipy.special.`**`y1p_zeros`**(*nt*, *complex=False*)

Compute nt zeros of Bessel derivative Y1'(z), and value at each zero.

The values are given by Y1(z1) at each z1 where Y1'(z1)=0.

Parameters **nt** : int

Number of zeros to return

complex : bool, default False

Set to False to return only the real zeros; set to True to return only the complex zeros with negative real part and positive imaginary part. Note that the complex conjugates of the latter are also zeros of the function, but are not returned by this routine.

Returns **z1pn** : ndarray

Location of nth zero of Y1'(z)

y1z1pn : ndarray

Value of derivative Y1(z1) for nth zero

References

[R539]

Faster versions of common Bessel Functions

`j0`(x)	Bessel function of the first kind of order 0.
`j1`(x)	Bessel function of the first kind of order 1.
`y0`(x)	Bessel function of the second kind of order 0.
`y1`(x)	Bessel function of the second kind of order 1.
`i0`(x)	Modified Bessel function of order 0.
`i0e`(x)	Exponentially scaled modified Bessel function of order 0.
`i1`(x)	Modified Bessel function of order 1.
`i1e`(x)	Exponentially scaled modified Bessel function of order 1.
`k0`(x)	Modified Bessel function of the second kind of order 0, K_0.

Continued on next page

Table 5.236 – continued from previous page

k0e(x)	Exponentially scaled modified Bessel function K of order 0
k1(x)	Modified Bessel function of the second kind of order 1, $K_1(x)$.
k1e(x)	Exponentially scaled modified Bessel function K of order 1

scipy.special.**j0**(*x*) = **<ufunc 'j0'>**

Bessel function of the first kind of order 0.

Parameters **x** : array_like

Argument (float).

Returns **J** : ndarray

Value of the Bessel function of the first kind of order 0 at *x*.

See also:

jv Bessel function of real order and complex argument.

spherical_jn
spherical Bessel functions.

Notes

The domain is divided into the intervals [0, 5] and (5, infinity). In the first interval the following rational approximation is used:

$$J_0(x) \approx (w - r_1^2)(w - r_2^2)\frac{P_3(w)}{Q_8(w)},$$

where $w = x^2$ and r_1, r_2 are the zeros of J_0, and P_3 and Q_8 are polynomials of degrees 3 and 8, respectively.

In the second interval, the Hankel asymptotic expansion is employed with two rational functions of degree 6/6 and 7/7.

This function is a wrapper for the Cephes *[R452]* routine *j0*. It should not to be confused with the spherical Bessel functions (see *spherical_jn*).

References

[R452]

scipy.special.**j1**(*x*) = **<ufunc 'j1'>**

Bessel function of the first kind of order 1.

Parameters **x** : array_like

Argument (float).

Returns **J** : ndarray

Value of the Bessel function of the first kind of order 1 at *x*.

See also:

jv

spherical_jn
spherical Bessel functions.

Notes

The domain is divided into the intervals [0, 8] and (8, infinity). In the first interval a 24 term Chebyshev expansion is used. In the second, the asymptotic trigonometric representation is employed using two rational functions of degree 5/5.

This function is a wrapper for the Cephes *[R453]* routine `j1`. It should not to be confused with the spherical Bessel functions (see `spherical_jn`).

References

[R453]

`scipy.special.y0(x)` = **<ufunc 'y0'>**

Bessel function of the second kind of order 0.

Parameters	**x** : array_like	Argument (float).
Returns	**Y** : ndarray	Value of the Bessel function of the second kind of order 0 at *x*.

See also:

`j0`, `yv`

Notes

The domain is divided into the intervals [0, 5] and (5, infinity). In the first interval a rational approximation $R(x)$ is employed to compute,

$$Y_0(x) = R(x) + \frac{2\log(x)J_0(x)}{\pi},$$

where J_0 is the Bessel function of the first kind of order 0.

In the second interval, the Hankel asymptotic expansion is employed with two rational functions of degree 6/6 and 7/7.

This function is a wrapper for the Cephes *[R535]* routine `y0`.

References

[R535]

`scipy.special.y1(x)` = **<ufunc 'y1'>**

Bessel function of the second kind of order 1.

Parameters	**x** : array_like	Argument (float).
Returns	**Y** : ndarray	Value of the Bessel function of the second kind of order 1 at *x*.

See also:

`j1`, `yn`, `yv`

Notes

The domain is divided into the intervals [0, 8] and (8, infinity). In the first interval a 25 term Chebyshev expansion is used, and computing J_1 (the Bessel function of the first kind) is required. In the second, the asymptotic trigonometric representation is employed using two rational functions of degree 5/5.

This function is a wrapper for the Cephes *[R537]* routine `y1`.

References

[R537]

`scipy.special.i0(x)` = **<ufunc 'i0'>**

Modified Bessel function of order 0.

Defined as,

$$I_0(x) = \sum_{k=0}^{\infty} \frac{(x^2/4)^k}{(k!)^2} = J_0(\imath x),$$

where J_0 is the Bessel function of the first kind of order 0.

Parameters **x** : array_like
Argument (float)

Returns **I** : ndarray
Value of the modified Bessel function of order 0 at *x*.

See also:

iv, *i0e*

Notes

The range is partitioned into the two intervals [0, 8] and (8, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R439]* routine *i0*.

References

[R439]

scipy.special.**i0e**(*x*) = <ufunc 'i0e'>

Exponentially scaled modified Bessel function of order 0.

Defined as:

```
i0e(x) = exp(-abs(x)) * i0(x).
```

Parameters **x** : array_like
Argument (float)

Returns **I** : ndarray
Value of the exponentially scaled modified Bessel function of order 0 at *x*.

See also:

iv, *i0*

Notes

The range is partitioned into the two intervals [0, 8] and (8, infinity). Chebyshev polynomial expansions are employed in each interval. The polynomial expansions used are the same as those in *i0*, but they are not multiplied by the dominant exponential factor.

This function is a wrapper for the Cephes *[R440]* routine *i0e*.

References

[R440]

scipy.special.**i1**(*x*) = <ufunc 'i1'>

Modified Bessel function of order 1.

Defined as,

$$I_1(x) = \frac{1}{2}x \sum_{k=0}^{\infty} \frac{(x^2/4)^k}{k!(k+1)!} = -\imath J_1(\imath x),$$

where J_1 is the Bessel function of the first kind of order 1.

Parameters **x** : array_like

Argument (float)

Returns **I** : ndarray

Value of the modified Bessel function of order 1 at *x*.

See also:

iv, *i1e*

Notes

The range is partitioned into the two intervals [0, 8] and (8, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R441]* routine *i1*.

References

[R441]

`scipy.special.i1e(x)` **= <ufunc 'i1e'>**

Exponentially scaled modified Bessel function of order 1.

Defined as:

```
i1e(x) = exp(-abs(x)) * i1(x)
```

Parameters **x** : array_like

Argument (float)

Returns **I** : ndarray

Value of the exponentially scaled modified Bessel function of order 1 at *x*.

See also:

iv, *i1*

Notes

The range is partitioned into the two intervals [0, 8] and (8, infinity). Chebyshev polynomial expansions are employed in each interval. The polynomial expansions used are the same as those in *i1*, but they are not multiplied by the dominant exponential factor.

This function is a wrapper for the Cephes *[R442]* routine *i1e*.

References

[R442]

`scipy.special.k0(x)` **= <ufunc 'k0'>**

Modified Bessel function of the second kind of order 0, K_0.

This function is also sometimes referred to as the modified Bessel function of the third kind of order 0.

Parameters **x** : array_like

Argument (float).

Returns **K** : ndarray

Value of the modified Bessel function K_0 at *x*.

See also:

kv, *k0e*

Notes

The range is partitioned into the two intervals [0, 2] and (2, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R463]* routine `k0`.

References

[R463]

`scipy.special.`**`k0e`**(*x*) = **<ufunc 'k0e'>**

Exponentially scaled modified Bessel function K of order 0

Defined as:

```
k0e(x) = exp(x) * k0(x).
```

Parameters **x** : array_like

Argument (float)

Returns **K** : ndarray

Value of the exponentially scaled modified Bessel function K of order 0 at *x*.

See also:

`kv`, `k0`

Notes

The range is partitioned into the two intervals [0, 2] and (2, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R464]* routine `k0e`.

References

[R464]

`scipy.special.`**`k1`**(*x*) = **<ufunc 'k1'>**

Modified Bessel function of the second kind of order 1, $K_1(x)$.

Parameters **x** : array_like

Argument (float)

Returns **K** : ndarray

Value of the modified Bessel function K of order 1 at *x*.

See also:

`kv`, `k1e`

Notes

The range is partitioned into the two intervals [0, 2] and (2, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R465]* routine `k1`.

References

[R465]

scipy.special.**k1e**(*x*) = <ufunc 'k1e'>
Exponentially scaled modified Bessel function K of order 1

Defined as:

```
k1e(x) = exp(x) * k1(x)
```

Parameters **x** : array_like
Argument (float)

Returns **K** : ndarray
Value of the exponentially scaled modified Bessel function K of order 1 at *x*.

See also:

kv, *k1*

Notes

The range is partitioned into the two intervals [0, 2] and (2, infinity). Chebyshev polynomial expansions are employed in each interval.

This function is a wrapper for the Cephes *[R466]* routine *k1e*.

References

[R466]

Integrals of Bessel Functions

itj0y0(x)	Integrals of Bessel functions of order 0
it2j0y0(x)	Integrals related to Bessel functions of order 0
iti0k0(x)	Integrals of modified Bessel functions of order 0
it2i0k0(x)	Integrals related to modified Bessel functions of order 0
besselpoly(a, lmb, nu)	Weighted integral of a Bessel function.

scipy.special.**itj0y0**(*x*) = <ufunc 'itj0y0'>
Integrals of Bessel functions of order 0

Returns simple integrals from 0 to *x* of the zeroth order Bessel functions *j0* and *y0*.

Returns ij0, iy0

scipy.special.**it2j0y0**(*x*) = <ufunc 'it2j0y0'>
Integrals related to Bessel functions of order 0

Returns ij0

```
integral((1-j0(t))/t, t=0..x)
```

iy0

```
integral(y0(t)/t, t=x..inf)
```

scipy.special.**iti0k0**(*x*) = <ufunc 'iti0k0'>
Integrals of modified Bessel functions of order 0

Returns simple integrals from 0 to *x* of the zeroth order modified Bessel functions *i0* and *k0*.

Returns ii0, ik0

scipy.special.**it2i0k0**(*x*) = <ufunc 'it2i0k0'>
Integrals related to modified Bessel functions of order 0

Returns ii0

```
integral((i0(t)-1)/t, t=0..x)
```

ik0

```
int(k0(t)/t, t=x..inf)
```

scipy.special.**besselpoly**(*a*, *lmb*, *nu*) = <ufunc 'besselpoly'>

Weighted integral of a Bessel function.

$$\int_0^1 x^\lambda J_\nu(2ax)\,dx$$

where J_ν is a Bessel function and $\lambda = lmb$, $\nu = nu$.

Derivatives of Bessel Functions

jvp(v, z[, n])	Compute nth derivative of Bessel function Jv(z) with respect to *z*.
yvp(v, z[, n])	Compute nth derivative of Bessel function Yv(z) with respect to *z*.
kvp(v, z[, n])	Compute nth derivative of real-order modified Bessel function Kv(z)
ivp(v, z[, n])	Compute nth derivative of modified Bessel function Iv(z) with respect to *z*.
h1vp(v, z[, n])	Compute nth derivative of Hankel function H1v(z) with respect to *z*.
h2vp(v, z[, n])	Compute nth derivative of Hankel function H2v(z) with respect to *z*.

scipy.special.**jvp**(*v*, *z*, *n=1*)

Compute nth derivative of Bessel function Jv(z) with respect to *z*.

Parameters **v** : float

Order of Bessel function

z : complex

Argument at which to evaluate the derivative

n : int, default 1

Order of derivative

Notes

The derivative is computed using the relation DLFM 10.6.7 *[R462]*.

References

[R461], *[R462]*

scipy.special.**yvp**(*v*, *z*, *n=1*)

Compute nth derivative of Bessel function Yv(z) with respect to *z*.

Parameters **v** : float

Order of Bessel function

z : complex

Argument at which to evaluate the derivative

n : int, default 1

Order of derivative

Notes

The derivative is computed using the relation DLFM 10.6.7 *[R546]*.

References

[R545], *[R546]*

`scipy.special.kvp(v, z, n=1)`

Compute nth derivative of real-order modified Bessel function Kv(z)

Kv(z) is the modified Bessel function of the second kind. Derivative is calculated with respect to *z*.

Parameters
- **v** : array_like of float
  Order of Bessel function
- **z** : array_like of complex
  Argument at which to evaluate the derivative
- **n** : int
  Order of derivative. Default is first derivative.

Returns
- **out** : ndarray
  The results

Notes

The derivative is computed using the relation DLFM 10.29.5 *[R479]*.

References

[R478], *[R479]*

Examples

Calculate multiple values at order 5:

```
>>> from scipy.special import kvp
>>> kvp(5, (1, 2, 3+5j))
array([-1849.0354+0.j    ,   -25.7735+0.j    ,    -0.0307+0.0875j])
```

Calculate for a single value at multiple orders:

```
>>> kvp((4, 4.5, 5), 1)
array([ -184.0309,  -568.9585, -1849.0354])
```

`scipy.special.ivp(v, z, n=1)`

Compute nth derivative of modified Bessel function Iv(z) with respect to *z*.

Parameters
- **v** : array_like of float
  Order of Bessel function
- **z** : array_like of complex
  Argument at which to evaluate the derivative
- **n** : int, default 1
  Order of derivative

Notes

The derivative is computed using the relation DLFM 10.29.5 *[R451]*.

References

[R450], *[R451]*

`scipy.special.h1vp(v, z, n=1)`

Compute nth derivative of Hankel function H1v(z) with respect to *z*.

Parameters
- **v** : float
  Order of Hankel function
- **z** : complex
  Argument at which to evaluate the derivative
- **n** : int, default 1
  Order of derivative

Notes

The derivative is computed using the relation DLFM 10.6.7 *[R432]*.

References

[R431], *[R432]*

`scipy.special.h2vp(v, z, n=1)`

Compute nth derivative of Hankel function H2v(z) with respect to *z*.

Parameters
- **v** : float
  Order of Hankel function
- **z** : complex
  Argument at which to evaluate the derivative
- **n** : int, default 1
  Order of derivative

Notes

The derivative is computed using the relation DLFM 10.6.7 *[R434]*.

References

[R433], *[R434]*

Spherical Bessel Functions

spherical_jn(n, z[, derivative])	Spherical Bessel function of the first kind or its derivative.
spherical_yn(n, z[, derivative])	Spherical Bessel function of the second kind or its derivative.
spherical_in(n, z[, derivative])	Modified spherical Bessel function of the first kind or its derivative.
spherical_kn(n, z[, derivative])	Modified spherical Bessel function of the second kind or its derivative.

`scipy.special.spherical_jn(n, z, derivative=False)`

Spherical Bessel function of the first kind or its derivative.

Defined as *[R524]*,

$$j_n(z) = \sqrt{\frac{\pi}{2z}} J_{n+1/2}(z),$$

where J_n is the Bessel function of the first kind.

Parameters
n : int, array_like
Order of the Bessel function (n >= 0).
z : complex or float, array_like
Argument of the Bessel function.
derivative : bool, optional
If True, the value of the derivative (rather than the function itself) is returned.

Returns
jn : ndarray

Notes

For real arguments greater than the order, the function is computed using the ascending recurrence *[R525]*. For small real or complex arguments, the definitional relation to the cylindrical Bessel function of the first kind is used.

The derivative is computed using the relations *[R526]*,

$$j_n' = j_{n-1} - \frac{n+1}{2} j_n.$$
$$j_0' = -j_1$$

New in version 0.18.0.

References

[R524], *[R525]*, *[R526]*

`scipy.special.`**`spherical_yn`**(*n*, *z*, *derivative=False*)

Spherical Bessel function of the second kind or its derivative.

Defined as *[R529]*,

$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+1/2}(z),$$

where Y_n is the Bessel function of the second kind.

Parameters
n : int, array_like
Order of the Bessel function (n >= 0).
z : complex or float, array_like
Argument of the Bessel function.
derivative : bool, optional
If True, the value of the derivative (rather than the function itself) is returned.

Returns
yn : ndarray

Notes

For real arguments, the function is computed using the ascending recurrence *[R530]*. For complex arguments, the definitional relation to the cylindrical Bessel function of the second kind is used.

The derivative is computed using the relations *[R531]*,

$$y_n' = y_{n-1} - \frac{n+1}{2} y_n.$$
$$y_0' = -y_1$$

New in version 0.18.0.

References

[R529], *[R530]*, *[R531]*

`scipy.special.`**`spherical_in`**(*n*, *z*, *derivative=False*)

Modified spherical Bessel function of the first kind or its derivative.

Defined as *[R522]*,

$$i_n(z) = \sqrt{\frac{\pi}{2z}} I_{n+1/2}(z),$$

where I_n is the modified Bessel function of the first kind.

Parameters
- **n** : int, array_like
  Order of the Bessel function (n >= 0).
- **z** : complex or float, array_like
  Argument of the Bessel function.
- **derivative** : bool, optional
  If True, the value of the derivative (rather than the function itself) is returned.

Returns
- **in** : ndarray

Notes

The function is computed using its definitional relation to the modified cylindrical Bessel function of the first kind.

The derivative is computed using the relations *[R523]*,

$$i'_n = i_{n-1} - \frac{n+1}{2} i_n.$$
$$i'_1 = i_0$$

New in version 0.18.0.

References

[R522], *[R523]*

`scipy.special.`**`spherical_kn`**(*n*, *z*, *derivative=False*)

Modified spherical Bessel function of the second kind or its derivative.

Defined as *[R527]*,

$$k_n(z) = \sqrt{\frac{\pi}{2z}} K_{n+1/2}(z),$$

where K_n is the modified Bessel function of the second kind.

Parameters
- **n** : int, array_like
  Order of the Bessel function (n >= 0).
- **z** : complex or float, array_like
  Argument of the Bessel function.
- **derivative** : bool, optional
  If True, the value of the derivative (rather than the function itself) is returned.

Returns
- **kn** : ndarray

Notes

The function is computed using its definitional relation to the modified cylindrical Bessel function of the second kind.

The derivative is computed using the relations *[R528]*,

$$k_n' = -k_{n-1} - \frac{n+1}{2} k_n.$$
$$k_0' = -k_1$$

New in version 0.18.0.

References

[R527], *[R528]*

Riccati-Bessel Functions

These are not universal functions:

riccati_jn(n, x)	Compute Ricatti-Bessel function of the first kind and its derivative.
riccati_yn(n, x)	Compute Ricatti-Bessel function of the second kind and its derivative.

`scipy.special.riccati_jn(n, x)`

Compute Ricatti-Bessel function of the first kind and its derivative.

The Ricatti-Bessel function of the first kind is defined as $xj_n(x)$, where j_n is the spherical Bessel function of the first kind of order n.

This function computes the value and first derivative of the Ricatti-Bessel function for all orders up to and including *n*.

Parameters

n : int
Maximum order of function to compute

x : float
Argument at which to evaluate

Returns

jn : ndarray
Value of j0(x), ..., jn(x)

jnp : ndarray
First derivative j0'(x), ..., jn'(x)

Notes

The computation is carried out via backward recurrence, using the relation DLMF 10.51.1 *[R511]*.

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R510]*.

References

[R510], *[R511]*

`scipy.special.riccati_yn(n, x)`

Compute Ricatti-Bessel function of the second kind and its derivative.

The Ricatti-Bessel function of the second kind is defined as $xy_n(x)$, where y_n is the spherical Bessel function of the second kind of order n.

This function computes the value and first derivative of the function for all orders up to and including *n*.

Parameters
- **n** : int
  Maximum order of function to compute
- **x** : float
  Argument at which to evaluate

Returns
- **yn** : ndarray
  Value of y0(x), ..., yn(x)
- **ynp** : ndarray
  First derivative y0'(x), ..., yn'(x)

Notes

The computation is carried out via ascending recurrence, using the relation DLMF 10.51.1 *[R513]*.

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R512]*.

References

[R512], *[R513]*

Struve Functions

`struve`(v, x)	Struve function.
`modstruve`(v, x)	Modified Struve function.
`itstruve0`(x)	Integral of the Struve function of order 0.
`it2struve0`(x)	Integral related to the Struve function of order 0.
`itmodstruve0`(x)	Integral of the modified Struve function of order 0.

`scipy.special.`**`struve`**(*v*, *x*) = **<ufunc 'struve'>**

Struve function.

Return the value of the Struve function of order *v* at *x*. The Struve function is defined as,

$$H_v(x) = (z/2)^{v+1} \sum_{n=0}^\infty \frac{(-1)^n (z/2)^{2n}}{\Gamma(n + \frac{3}{2}) \Gamma(n + v + \frac{3}{2})},$$

where Γ is the gamma function.

Parameters
- **v** : array_like
  Order of the Struve function (float).
- **x** : array_like
  Argument of the Struve function (float; must be positive unless *v* is an integer).

Returns
- **H** : ndarray
  Value of the Struve function of order *v* at *x*.

See also:

`modstruve`

Notes

Three methods discussed in *[R532]* are used to evaluate the Struve function:
- power series
- expansion in Bessel functions (if $|z| < |v| + 20$)
- asymptotic large-z expansion (if $z \geq 0.7v + 12$)

Rounding errors are estimated based on the largest terms in the sums, and the result associated with the smallest error is returned.

References

[R532]

`scipy.special.modstruve(v, x)` **= <ufunc 'modstruve'>**

Modified Struve function.

Return the value of the modified Struve function of order *v* at *x*. The modified Struve function is defined as,

$$L_v(x) = -\imath \exp(-\pi\imath v/2) H_v(x),$$

where H_v is the Struve function.

Parameters
- **v** : array_like
  Order of the modified Struve function (float).
- **x** : array_like
  Argument of the Struve function (float; must be positive unless *v* is an integer).

Returns
- **L** : ndarray
  Value of the modified Struve function of order *v* at *x*.

See also:

struve

Notes

Three methods discussed in *[R493]* are used to evaluate the function:

- power series
- expansion in Bessel functions (if $|z| < |v| + 20$)
- asymptotic large-z expansion (if $z \geq 0.7v + 12$)

Rounding errors are estimated based on the largest terms in the sums, and the result associated with the smallest error is returned.

References

[R493]

`scipy.special.itstruve0(x)` **= <ufunc 'itstruve0'>**

Integral of the Struve function of order 0.

$$I = \int_0^x H_0(t)\,dt$$

Parameters
- **x** : array_like
  Upper limit of integration (float).

Returns
- **I** : ndarray
  The integral of H_0 from 0 to *x*.

See also:

struve

Notes

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R446]*.

References

[R446]

`scipy.special.it2struve0`(*x*) = **<ufunc 'it2struve0'>**

Integral related to the Struve function of order 0.

Returns the integral,

$$\int_x^\infty \frac{H_0(t)}{t}\,dt$$

where H_0 is the Struve function of order 0.

Parameters **x** : array_like

Lower limit of integration.

Returns **I** : ndarray

The value of the integral.

See also:

struve

Notes

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R443]*.

References

[R443]

`scipy.special.itmodstruve0`(*x*) = **<ufunc 'itmodstruve0'>**

Integral of the modified Struve function of order 0.

$$I = \int_0^x L_0(t)\,dt$$

Parameters **x** : array_like

Upper limit of integration (float).

Returns **I** : ndarray

The integral of L_0 from 0 to x.

Notes

Wrapper for a Fortran routine created by Shanjie Zhang and Jianming Jin *[R445]*.

References

[R445]

Raw Statistical Functions

See also:

scipy.stats: Friendly versions of these functions.

bdtr(k, n, p)	Binomial distribution cumulative distribution function.
bdtrc(k, n, p)	Binomial distribution survival function.
bdtri(k, n, y)	Inverse function to *bdtr* with respect to *p*.
bdtrik(y, n, p)	Inverse function to *bdtr* with respect to *k*.
bdtrin(k, y, p)	Inverse function to *bdtr* with respect to *n*.
btdtr(a, b, x)	Cumulative density function of the beta distribution.
btdtri(a, b, p)	The *p*-th quantile of the beta distribution.

Continued on next page

Table 5.242 – continued from previous page

btdtria(p, b, x)	Inverse of `btdtr` with respect to *a*.
btdtrib(a, p, x)	Inverse of `btdtr` with respect to *b*.
fdtr(dfn, dfd, x)	F cumulative distribution function.
fdtrc(dfn, dfd, x)	F survival function.
fdtri(dfn, dfd, p)	The *p*-th quantile of the F-distribution.
fdtridfd(dfn, p, x)	Inverse to `fdtr` vs dfd
gdtr(a, b, x)	Gamma distribution cumulative density function.
gdtrc(a, b, x)	Gamma distribution survival function.
gdtria(p, b, x[, out])	Inverse of `gdtr` vs a.
gdtrib(a, p, x[, out])	Inverse of `gdtr` vs b.
gdtrix(a, b, p[, out])	Inverse of `gdtr` vs x.
nbdtr(k, n, p)	Negative binomial cumulative distribution function.
nbdtrc(k, n, p)	Negative binomial survival function.
nbdtri(k, n, y)	Inverse of `nbdtr` vs *p*.
nbdtrik(y, n, p)	Inverse of `nbdtr` vs *k*.
nbdtrin(k, y, p)	Inverse of `nbdtr` vs *n*.
ncfdtr(dfn, dfd, nc, f)	Cumulative distribution function of the non-central F distribution.
ncfdtridfd(p, f, dfn, nc)	Calculate degrees of freedom (denominator) for the noncentral F-distribution.
ncfdtridfn(p, f, dfd, nc)	Calculate degrees of freedom (numerator) for the noncentral F-distribution.
ncfdtri(p, dfn, dfd, nc)	Inverse cumulative distribution function of the non-central F distribution.
ncfdtrinc(p, f, dfn, dfd)	Calculate non-centrality parameter for non-central F distribution.
nctdtr(df, nc, t)	Cumulative distribution function of the non-central *t* distribution.
nctdtridf(p, nc, t)	Calculate degrees of freedom for non-central t distribution.
nctdtrit(df, nc, p)	Inverse cumulative distribution function of the non-central t distribution.
nctdtrinc(df, p, t)	Calculate non-centrality parameter for non-central t distribution.
nrdtrimn(p, x, std)	Calculate mean of normal distribution given other params.
nrdtrisd(p, x, mn)	Calculate standard deviation of normal distribution given other params.
pdtr(k, m)	Poisson cumulative distribution function
pdtrc(k, m)	Poisson survival function
pdtri(k, y)	Inverse to `pdtr` vs m
pdtrik(p, m)	Inverse to `pdtr` vs k
stdtr(df, t)	Student t distribution cumulative density function
stdtridf(p, t)	Inverse of `stdtr` vs df
stdtrit(df, p)	Inverse of `stdtr` vs *t*
chdtr(v, x)	Chi square cumulative distribution function
chdtrc(v, x)	Chi square survival function
chdtri(v, p)	Inverse to `chdtrc`
chdtriv(p, x)	Inverse to `chdtr` vs *v*
ndtr(x)	Gaussian cumulative distribution function.
log_ndtr(x)	Logarithm of Gaussian cumulative distribution function.
ndtri(y)	Inverse of `ndtr` vs x

Continued on next page

Table 5.242 – continued from previous page

chndtr(x, df, nc)	Non-central chi square cumulative distribution function
chndtridf(x, p, nc)	Inverse to *chndtr* vs *df*
chndtrinc(x, df, p)	Inverse to *chndtr* vs *nc*
chndtrix(p, df, nc)	Inverse to *chndtr* vs *x*
smirnov(n, e)	Kolmogorov-Smirnov complementary cumulative distribution function
smirnovi(n, y)	Inverse to *smirnov*
kolmogorov(y)	Complementary cumulative distribution function of Kolmogorov distribution
kolmogi(p)	Inverse function to kolmogorov
tklmbda(x, lmbda)	Tukey-Lambda cumulative distribution function
logit(x)	Logit ufunc for ndarrays.
expit(x)	Expit ufunc for ndarrays.
boxcox(x, lmbda)	Compute the Box-Cox transformation.
boxcox1p(x, lmbda)	Compute the Box-Cox transformation of 1 + *x*.
inv_boxcox(y, lmbda)	Compute the inverse of the Box-Cox transformation.
inv_boxcox1p(y, lmbda)	Compute the inverse of the Box-Cox transformation.

scipy.special.**bdtr**(*k*, *n*, *p*) = **<ufunc 'bdtr'>**

Binomial distribution cumulative distribution function.

Sum of the terms 0 through *k* of the Binomial probability density.

$$\mathrm{bdtr}(k,n,p) = \sum_{j=0}^{k} \binom{n}{j} p^j (1-p)^{n-j}$$

Parameters

k : array_like
Number of successes (int).

n : array_like
Number of events (int).

p : array_like
Probability of success in a single event (float).

Returns

y : ndarray
Probability of *k* or fewer successes in *n* independent events with success probabilities of *p*.

Notes

The terms are not summed directly; instead the regularized incomplete beta function is employed, according to the formula,

$$\mathrm{bdtr}(k,n,p) = I_{1-p}(n-k, k+1).$$

Wrapper for the Cephes *[R371]* routine *bdtr*.

References

[R371]

scipy.special.**bdtrc**(*k*, *n*, *p*) = **<ufunc 'bdtrc'>**

Binomial distribution survival function.

Sum of the terms *k + 1* through *n* of the binomial probability density,

$$\mathrm{bdtrc}(k,n,p) = \sum_{j=k+1}^{n} \binom{n}{j} p^j (1-p)^{n-j}$$

Parameters **k** : array_like

Number of successes (int).

n : array_like

Number of events (int)

p : array_like

Probability of success in a single event.

Returns **y** : ndarray

Probability of *k + 1* or more successes in *n* independent events with success probabilities of *p*.

See also:

bdtr, *betainc*

Notes

The terms are not summed directly; instead the regularized incomplete beta function is employed, according to the formula,

$$\mathrm{bdtrc}(k, n, p) = I_p(k + 1, n - k).$$

Wrapper for the Cephes *[R372]* routine *bdtrc*.

References

[R372]

`scipy.special.bdtri`(*k*, *n*, *y*) **= <ufunc 'bdtri'>**

Inverse function to *bdtr* with respect to *p*.

Finds the event probability *p* such that the sum of the terms 0 through *k* of the binomial probability density is equal to the given cumulative probability *y*.

Parameters **k** : array_like

Number of successes (float).

n : array_like

Number of events (float)

y : array_like

Cumulative probability (probability of *k* or fewer successes in *n* events).

Returns **p** : ndarray

The event probability such that *bdtr(k, n, p) = y*.

See also:

bdtr, *betaincinv*

Notes

The computation is carried out using the inverse beta integral function and the relation,:

```
1 - p = betaincinv(n - k, k + 1, y).
```

Wrapper for the Cephes *[R373]* routine *bdtri*.

References

[R373]

`scipy.special.bdtrik`(*y*, *n*, *p*) **= <ufunc 'bdtrik'>**

Inverse function to *bdtr* with respect to *k*.

Finds the number of successes *k* such that the sum of the terms 0 through *k* of the Binomial probability density for *n* events with probability *p* is equal to the given cumulative probability *y*.

Parameters

y : array_like
: Cumulative probability (probability of *k* or fewer successes in *n* events).

n : array_like
: Number of events (float).

p : array_like
: Success probability (float).

Returns

k : ndarray
: The number of successes *k* such that *bdtr(k, n, p) = y*.

See also:

`bdtr`

Notes

Formula 26.5.24 of *[R374]* is used to reduce the binomial distribution to the cumulative incomplete beta distribution.

Computation of *k* involves a seach for a value that produces the desired value of *y*. The search relies on the monotinicity of *y* with *k*.

Wrapper for the CDFLIB *[R375]* Fortran routine *cdfbin*.

References

[R374], *[R375]*

`scipy.special.`**`bdtrin`**(*k*, *y*, *p*) = **<ufunc 'bdtrin'>**

Inverse function to `bdtr` with respect to *n*.

Finds the number of events *n* such that the sum of the terms 0 through *k* of the Binomial probability density for events with probability *p* is equal to the given cumulative probability *y*.

Parameters

k : array_like
: Number of successes (float).

y : array_like
: Cumulative probability (probability of *k* or fewer successes in *n* events).

p : array_like
: Success probability (float).

Returns

n : ndarray
: The number of events *n* such that *bdtr(k, n, p) = y*.

See also:

`bdtr`

Notes

Formula 26.5.24 of *[R376]* is used to reduce the binomial distribution to the cumulative incomplete beta distribution.

Computation of *n* involves a seach for a value that produces the desired value of *y*. The search relies on the monotinicity of *y* with *n*.

Wrapper for the CDFLIB *[R377]* Fortran routine *cdfbin*.

References

[R376], *[R377]*

`scipy.special.`**`btdtr`**(*a*, *b*, *x*) = **<ufunc 'btdtr'>**

Cumulative density function of the beta distribution.

Returns the integral from zero to *x* of the beta probability density function,

$$I = \int_0^x \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} t^{a-1}(1-t)^{b-1}\,dt$$

where Γ is the gamma function.

Parameters
- **a** : array_like
  Shape parameter (a > 0).
- **b** : array_like
  Shape parameter (b > 0).
- **x** : array_like
  Upper limit of integration, in [0, 1].

Returns
- **I** : ndarray
  Cumulative density function of the beta distribution with parameters *a* and *b* at *x*.

See also:

`betainc`

Notes

This function is identical to the incomplete beta integral function `betainc`.

Wrapper for the Cephes *[R384]* routine `btdtr`.

References

[R384]

`scipy.special.`**`btdtri`**(*a*, *b*, *p*) **= <ufunc 'btdtri'>**

The *p*-th quantile of the beta distribution.

This function is the inverse of the beta cumulative distribution function, `btdtr`, returning the value of *x* for which *btdtr(a, b, x) = p*, or

$$p = \int_0^x \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} t^{a-1}(1-t)^{b-1}\,dt$$

Parameters
- **a** : array_like
  Shape parameter ($a > 0$).
- **b** : array_like
  Shape parameter ($b > 0$).
- **p** : array_like
  Cumulative probability, in [0, 1].

Returns
- **x** : ndarray
  The quantile corresponding to *p*.

See also:

`betaincinv`, `btdtr`

Notes

The value of *x* is found by interval halving or Newton iterations.

Wrapper for the Cephes *[R385]* routine *incbi*, which solves the equivalent problem of finding the inverse of the incomplete beta integral.

References

[R385]

scipy.special.**btdtria**(p, b, x) = **<ufunc 'btdtria'>**

Inverse of `btdtr` with respect to a.

This is the inverse of the beta cumulative distribution function, `btdtr`, considered as a function of a, returning the value of a for which *btdtr(a, b, x) = p*, or

$$p = \int_0^x \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} t^{a-1}(1-t)^{b-1}\,dt$$

Parameters

p : array_like
: Cumulative probability, in [0, 1].

b : array_like
: Shape parameter ($b > 0$).

x : array_like
: The quantile, in [0, 1].

Returns

a : ndarray
: The value of the shape parameter a such that *btdtr(a, b, x) = p*.

See also:

btdtr Cumulative density function of the beta distribution.
btdtri Inverse with respect to x.
btdtrib Inverse with respect to b.

Notes

Wrapper for the CDFLIB *[R386]* Fortran routine *cdfbet*.

The cumulative distribution function p is computed using a routine by DiDinato and Morris *[R387]*. Computation of a involves a seach for a value that produces the desired value of p. The search relies on the monotinicity of p with a.

References

[R386], *[R387]*

scipy.special.**btdtrib**(a, p, x) = **<ufunc 'btdtrib'>**

Inverse of `btdtr` with respect to b.

This is the inverse of the beta cumulative distribution function, `btdtr`, considered as a function of b, returning the value of b for which *btdtr(a, b, x) = p*, or

$$p = \int_0^x \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} t^{a-1}(1-t)^{b-1}\,dt$$

Parameters

a : array_like
: Shape parameter ($a > 0$).

p : array_like
: Cumulative probability, in [0, 1].

x : array_like
: The quantile, in [0, 1].

Returns

b : ndarray
: The value of the shape parameter b such that *btdtr(a, b, x) = p*.

See also:

btdtr Cumulative density function of the beta distribution.
btdtri Inverse with respect to x.
btdtria Inverse with respect to a.

Notes

Wrapper for the CDFLIB *[R388]* Fortran routine *cdfbet*.

The cumulative distribution function *p* is computed using a routine by DiDinato and Morris *[R389]*. Computation of *b* involves a seach for a value that produces the desired value of *p*. The search relies on the monotinicity of *p* with *b*.

References

[R388], *[R389]*

`scipy.special.fdtr`(*dfn*, *dfd*, *x*) **= <ufunc 'fdtr'>**

F cumulative distribution function.

Returns the value of the cumulative density function of the F-distribution, also known as Snedecor's F-distribution or the Fisher-Snedecor distribution.

The F-distribution with parameters d_n and d_d is the distribution of the random variable,

$$X = \frac{U_n/d_n}{U_d/d_d},$$

where U_n and U_d are random variables distributed χ^2, with d_n and d_d degrees of freedom, respectively.

Parameters
- **dfn** : array_like
  First parameter (positive float).
- **dfd** : array_like
  Second parameter (positive float).
- **x** : array_like
  Argument (nonnegative float).

Returns
- **y** : ndarray
  The CDF of the F-distribution with parameters *dfn* and *dfd* at *x*.

Notes

The regularized incomplete beta function is used, according to the formula,

$$F(d_n, d_d; x) = I_{xd_n/(d_d + xd_n)}(d_n/2, d_d/2).$$

Wrapper for the Cephes *[R415]* routine `fdtr`.

References

[R415]

`scipy.special.fdtrc`(*dfn*, *dfd*, *x*) **= <ufunc 'fdtrc'>**

F survival function.

Returns the complemented F-distribution function (the integral of the density from *x* to infinity).

Parameters
- **dfn** : array_like
  First parameter (positive float).
- **dfd** : array_like
  Second parameter (positive float).
- **x** : array_like
  Argument (nonnegative float).

Returns
- **y** : ndarray
  The complemented F-distribution function with parameters *dfn* and *dfd* at *x*.

See also:

fdtr

Notes

The regularized incomplete beta function is used, according to the formula,

$$F(d_n, d_d; x) = I_{d_d/(d_d + xd_n)}(d_d/2, d_n/2).$$

Wrapper for the Cephes *[R416]* routine `fdtrc`.

References

[R416]

`scipy.special.`**`fdtri`**(*dfn*, *dfd*, *p*) = **<ufunc 'fdtri'>**

The *p*-th quantile of the F-distribution.

This function is the inverse of the F-distribution CDF, `fdtr`, returning the *x* such that *fdtr(dfn, dfd, x) = p*.

Parameters **dfn** : array_like
First parameter (positive float).
dfd : array_like
Second parameter (positive float).
p : array_like
Cumulative probability, in [0, 1].

Returns **x** : ndarray
The quantile corresponding to *p*.

Notes

The computation is carried out using the relation to the inverse regularized beta function, $I^{-1}_x(a, b)$. Let $z = I^{-1}_p(d_d/2, d_n/2)$. Then,

$$x = \frac{d_d(1 - z)}{d_n z}.$$

If *p* is such that $x < 0.5$, the following relation is used instead for improved stability: let $z' = I^{-1}_{1-p}(d_n/2, d_d/2)$. Then,

$$x = \frac{d_d z'}{d_n(1 - z')}.$$

Wrapper for the Cephes *[R417]* routine `fdtri`.

References

[R417]

`scipy.special.`**`fdtridfd`**(*dfn*, *p*, *x*) = **<ufunc 'fdtridfd'>**

Inverse to `fdtr` vs dfd

Finds the F density argument dfd such that `fdtr(dfn, dfd, x) == p`.

`scipy.special.`**`gdtr`**(*a*, *b*, *x*) = **<ufunc 'gdtr'>**

Gamma distribution cumulative density function.

Returns the integral from zero to *x* of the gamma probability density function,

$$F = \int_0^x \frac{a^b}{\Gamma(b)} t^{b-1} e^{-at}\,dt,$$

where Γ is the gamma function.

Parameters **a** : array_like

The rate parameter of the gamma distribution, sometimes denoted β (float). It is also the reciprocal of the scale parameter θ.

b : array_like

The shape parameter of the gamma distribution, sometimes denoted α (float).

x : array_like

The quantile (upper limit of integration; float).

Returns **F** : ndarray

The CDF of the gamma distribution with parameters *a* and *b* evaluated at *x*.

See also:

gdtrc 1 - CDF of the gamma distribution.

Notes

The evaluation is carried out using the relation to the incomplete gamma integral (regularized gamma function).

Wrapper for the Cephes *[R423]* routine `gdtr`.

References

[R423]

`scipy.special.`**`gdtrc`**(a, b, x) **= <ufunc 'gdtrc'>**

Gamma distribution survival function.

Integral from *x* to infinity of the gamma probability density function,

$$F = \int_x^\infty \frac{a^b}{\Gamma(b)} t^{b-1} e^{-at}\,dt,$$

where Γ is the gamma function.

Parameters **a** : array_like

The rate parameter of the gamma distribution, sometimes denoted β (float). It is also the reciprocal of the scale parameter θ.

b : array_like

The shape parameter of the gamma distribution, sometimes denoted α (float).

x : array_like

The quantile (lower limit of integration; float).

Returns **F** : ndarray

The survival function of the gamma distribution with parameters *a* and *b* evaluated at *x*.

See also:

`gdtr`, `gdtri`

Notes

The evaluation is carried out using the relation to the incomplete gamma integral (regularized gamma function).

Wrapper for the Cephes *[R424]* routine `gdtrc`.

References

[R424]

`scipy.special.`**`gdtria`**$(p, b, x, out=None)$ **= <ufunc 'gdtria'>**

Inverse of `gdtr` vs a.

Returns the inverse with respect to the parameter *a* of `p = gdtr(a, b, x)`, the cumulative distribution function of the gamma distribution.

Parameters **p** : array_like

Probability values.

b : array_like

b parameter values of *gdtr(a, b, x)*. *b* is the "shape" parameter of the gamma distribution.

x : array_like

Nonnegative real values, from the domain of the gamma distribution.

out : ndarray, optional

If a fourth argument is given, it must be a numpy.ndarray whose size matches the broadcast result of *a*, *b* and *x*. *out* is then the array returned by the function.

Returns **a** : ndarray

Values of the *a* parameter such that *p = gdtr(a, b, x)*. *1/a* is the "scale" parameter of the gamma distribution.

See also:

gdtr	CDF of the gamma distribution.
gdtrib	Inverse with respect to *b* of *gdtr(a, b, x)*.
gdtrix	Inverse with respect to *x* of *gdtr(a, b, x)*.

Notes

Wrapper for the CDFLIB *[R425]* Fortran routine *cdfgam*.

The cumulative distribution function *p* is computed using a routine by DiDinato and Morris *[R426]*. Computation of *a* involves a seach for a value that produces the desired value of *p*. The search relies on the monotinicity of *p* with *a*.

References

[R425], *[R426]*

Examples

First evaluate `gdtr`.

```
>>> from scipy.special import gdtr, gdtria
>>> p = gdtr(1.2, 3.4, 5.6)
>>> print(p)
0.94378087442
```

Verify the inverse.

```
>>> gdtria(p, 3.4, 5.6)
1.2
```

`scipy.special.`**gdtrib**(*a*, *p*, *x*, *out=None*) **= <ufunc 'gdtrib'>**

Inverse of `gdtr` vs b.

Returns the inverse with respect to the parameter *b* of `p = gdtr(a, b, x)`, the cumulative distribution function of the gamma distribution.

Parameters **a** : array_like

a parameter values of *gdtr(a, b, x)*. *1/a* is the "scale" parameter of the gamma distribution.

p : array_like

Probability values.

x : array_like
    Nonnegative real values, from the domain of the gamma distribution.

out : ndarray, optional
    If a fourth argument is given, it must be a numpy.ndarray whose size matches the broadcast result of *a*, *b* and *x*. *out* is then the array returned by the function.

Returns **b** : ndarray
    Values of the *b* parameter such that *p = gdtr(a, b, x)*. *b* is the "shape" parameter of the gamma distribution.

See also:

gdtr CDF of the gamma distribution.
gdtria Inverse with respect to *a* of *gdtr(a, b, x)*.
gdtrix Inverse with respect to *x* of *gdtr(a, b, x)*.

Notes

Wrapper for the CDFLIB *[R427]* Fortran routine *cdfgam*.

The cumulative distribution function *p* is computed using a routine by DiDinato and Morris *[R428]*. Computation of *b* involves a seach for a value that produces the desired value of *p*. The search relies on the monotinicity of *p* with *b*.

References

[R427], *[R428]*

Examples

First evaluate `gdtr`.

```
>>> from scipy.special import gdtr, gdtrib
>>> p = gdtr(1.2, 3.4, 5.6)
>>> print(p)
0.94378087442
```

Verify the inverse.

```
>>> gdtrib(1.2, p, 5.6)
3.3999999999723882
```

`scipy.special.`**`gdtrix`**(*a*, *b*, *p*, *out=None*) **= <ufunc 'gdtrix'>**

Inverse of `gdtr` vs x.

Returns the inverse with respect to the parameter *x* of `p = gdtr(a, b, x)`, the cumulative distribution function of the gamma distribution. This is also known as the p'th quantile of the distribution.

Parameters **a** : array_like
    a parameter values of *gdtr(a, b, x)*. *1/a* is the "scale" parameter of the gamma distribution.

b : array_like
    b parameter values of *gdtr(a, b, x)*. *b* is the "shape" parameter of the gamma distribution.

p : array_like
    Probability values.

out : ndarray, optional

If a fourth argument is given, it must be a numpy.ndarray whose size matches the broadcast result of a, b and x. *out* is then the array returned by the function.

Returns **x** : ndarray

Values of the x parameter such that p = *gdtr(a, b, x)*.

See also:

gdtr CDF of the gamma distribution.
gdtria Inverse with respect to a of *gdtr(a, b, x)*.
gdtrib Inverse with respect to b of *gdtr(a, b, x)*.

Notes

Wrapper for the CDFLIB *[R429]* Fortran routine *cdfgam*.

The cumulative distribution function p is computed using a routine by DiDinato and Morris *[R430]*. Computation of x involves a seach for a value that produces the desired value of p. The search relies on the monotinicity of p with x.

References

[R429], *[R430]*

Examples

First evaluate `gdtr`.

```
>>> from scipy.special import gdtr, gdtrix
>>> p = gdtr(1.2, 3.4, 5.6)
>>> print(p)
0.94378087442
```

Verify the inverse.

```
>>> gdtrix(1.2, 3.4, p)
5.5999999999999996
```

`scipy.special.`**`nbdtr`**(*k*, *n*, *p*) **= <ufunc 'nbdtr'>**

Negative binomial cumulative distribution function.

Returns the sum of the terms 0 through k of the negative binomial distribution probability mass function,

$$F = \sum_{j=0}^{k} \binom{n + j - 1}{j} p^n (1 - p)^j.$$

In a sequence of Bernoulli trials with individual success probabilities p, this is the probability that k or fewer failures precede the nth success.

Parameters **k** : array_like

The maximum number of allowed failures (nonnegative int).

n : array_like

The target number of successes (positive int).

p : array_like

Probability of success in a single event (float).

Returns **F** : ndarray

The probability of k or fewer failures before n successes in a sequence of events with individual success probability p.

See also:

nbdtrc

Notes

If floating point values are passed for *k* or *n*, they will be truncated to integers.

The terms are not summed directly; instead the regularized incomplete beta function is employed, according to the formula,

$$\mathrm{nbdtr}(k, n, p) = I_p(n, k + 1).$$

Wrapper for the Cephes *[R494]* routine `nbdtr`.

References

[R494]

`scipy.special.nbdtrc(k, n, p) = <ufunc 'nbdtrc'>`

Negative binomial survival function.

Returns the sum of the terms *k + 1* to infinity of the negative binomial distribution probability mass function,

$$F = \sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^n (1-p)^j.$$

In a sequence of Bernoulli trials with individual success probabilities *p*, this is the probability that more than *k* failures precede the nth success.

Parameters

- **k** : array_like
  The maximum number of allowed failures (nonnegative int).
- **n** : array_like
  The target number of successes (positive int).
- **p** : array_like
  Probability of success in a single event (float).

Returns

- **F** : ndarray
  The probability of *k + 1* or more failures before *n* successes in a sequence of events with individual success probability *p*.

Notes

If floating point values are passed for *k* or *n*, they will be truncated to integers.

The terms are not summed directly; instead the regularized incomplete beta function is employed, according to the formula,

$$\mathrm{nbdtrc}(k, n, p) = I_{1-p}(k + 1, n).$$

Wrapper for the Cephes *[R495]* routine `nbdtrc`.

References

[R495]

`scipy.special.nbdtri(k, n, y) = <ufunc 'nbdtri'>`

Inverse of `nbdtr` vs *p*.

Returns the inverse with respect to the parameter *p* of *y = nbdtr(k, n, p)*, the negative binomial cumulative distribution function.

Parameters

- **k** : array_like
  The maximum number of allowed failures (nonnegative int).
- **n** : array_like
  The target number of successes (positive int).

y : array_like
The probability of k or fewer failures before n successes (float).

Returns **p** : ndarray
Probability of success in a single event (float) such that *nbdtr(k, n, p) = y*.

See also:

nbdtr Cumulative distribution function of the negative binomial.
nbdtrik Inverse with respect to k of *nbdtr(k, n, p)*.
nbdtrin Inverse with respect to n of *nbdtr(k, n, p)*.

Notes

Wrapper for the Cephes *[R496]* routine `nbdtri`.

References

[R496]

`scipy.special.`**nbdtrik**(y, n, p) **= <ufunc 'nbdtrik'>**

Inverse of `nbdtr` vs k.

Returns the inverse with respect to the parameter k of y = *nbdtr(k, n, p)*, the negative binomial cumulative distribution function.

Parameters **y** : array_like
The probability of k or fewer failures before n successes (float).

n : array_like
The target number of successes (positive int).

p : array_like
Probability of success in a single event (float).

Returns **k** : ndarray
The maximum number of allowed failures such that *nbdtr(k, n, p) = y*.

See also:

nbdtr Cumulative distribution function of the negative binomial.
nbdtri Inverse with respect to p of *nbdtr(k, n, p)*.
nbdtrin Inverse with respect to n of *nbdtr(k, n, p)*.

Notes

Wrapper for the CDFLIB *[R497]* Fortran routine *cdfnbn*.

Formula 26.5.26 of *[R498]*,

$$\sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^n (1-p)^j = I_{1-p}(k+1, n),$$

is used to reduce calculation of the cumulative distribution function to that of a regularized incomplete beta I.

Computation of k involves a seach for a value that produces the desired value of y. The search relies on the monotinicity of y with k.

References

[R497], *[R498]*

`scipy.special.`**nbdtrin**(k, y, p) **= <ufunc 'nbdtrin'>**

Inverse of `nbdtr` vs n.

Returns the inverse with respect to the parameter n of y = *nbdtr(k, n, p)*, the negative binomial cumulative distribution function.

Parameters **k** : array_like
The maximum number of allowed failures (nonnegative int).
y : array_like
The probability of *k* or fewer failures before *n* successes (float).
p : array_like
Probability of success in a single event (float).

Returns **n** : ndarray
The number of successes *n* such that *nbdtr(k, n, p) = y*.

See also:

nbdtr Cumulative distribution function of the negative binomial.
nbdtri Inverse with respect to *p* of *nbdtr(k, n, p)*.
nbdtrik Inverse with respect to *k* of *nbdtr(k, n, p)*.

Notes

Wrapper for the CDFLIB *[R499]* Fortran routine *cdfnbn*.

Formula 26.5.26 of *[R500]*,

$$\sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^n (1-p)^j = I_{1-p}(k+1, n),$$

is used to reduce calculation of the cumulative distribution function to that of a regularized incomplete beta I.

Computation of *n* involves a seach for a value that produces the desired value of *y*. The search relies on the monotinicity of *y* with *n*.

References

[R499], *[R500]*

`scipy.special.ncfdtr`(*dfn*, *dfd*, *nc*, *f*) = **<ufunc 'ncfdtr'>**
Cumulative distribution function of the non-central F distribution.

The non-central F describes the distribution of,

$$Z = \frac{X/d_n}{Y/d_d}$$

where X and Y are independently distributed, with X distributed non-central χ^2 with noncentrality parameter *nc* and d_n degrees of freedom, and Y distributed χ^2 with d_d degrees of freedom.

Parameters **dfn** : array_like
Degrees of freedom of the numerator sum of squares. Range (0, inf).
dfd : array_like
Degrees of freedom of the denominator sum of squares. Range (0, inf).
nc : array_like
Noncentrality parameter. Should be in range (0, 1e4).
f : array_like
Quantiles, i.e. the upper limit of integration.

Returns **cdf** : float or ndarray
The calculated CDF. If all inputs are scalar, the return will be a float. Otherwise it will be an array.

See also:

ncdfdtri Inverse CDF (iCDF) of the non-central F distribution.
ncdfdtridfd
Calculate dfd, given CDF and iCDF values.

ncdfdtridfn Calculate dfn, given CDF and iCDF values.

ncdfdtrinc Calculate noncentrality parameter, given CDF, iCDF, dfn, dfd.

Notes

Wrapper for the CDFLIB *[R501]* Fortran routine *cdffnc*.

The cumulative distribution function is computed using Formula 26.6.20 of *[R502]*:

$$F(d_n, d_d, n_c, f) = \sum_{j=0}^{\infty} e^{-n_c/2} \frac{(n_c/2)^j}{j!} I_x(\frac{d_n}{2} + j, \frac{d_d}{2}),$$

where I is the regularized incomplete beta function, and $x = f d_n/(f d_n + d_d)$.

The computation time required for this routine is proportional to the noncentrality parameter *nc*. Very large values of this parameter can consume immense computer resources. This is why the search range is bounded by 10,000.

References

[R501], *[R502]*

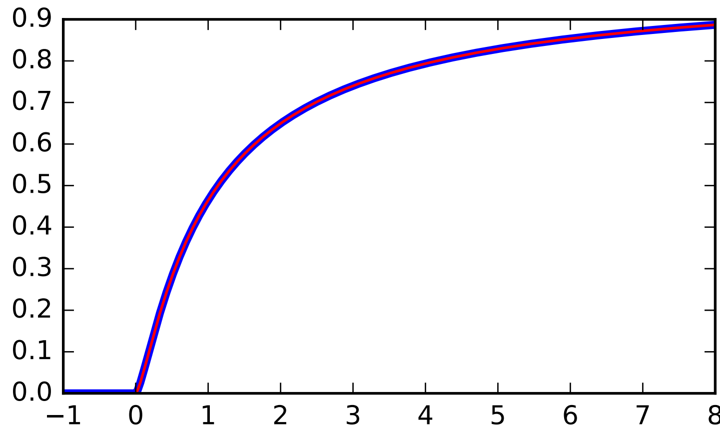
Examples

```
>>> from scipy import special
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

Plot the CDF of the non-central F distribution, for nc=0. Compare with the F-distribution from scipy.stats:

```
>>> x = np.linspace(-1, 8, num=500)
>>> dfn = 3
>>> dfd = 2
>>> ncf_stats = stats.f.cdf(x, dfn, dfd)
>>> ncf_special = special.ncfdtr(dfn, dfd, 0, x)
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, ncf_stats, 'b-', lw=3)
>>> ax.plot(x, ncf_special, 'r-')
>>> plt.show()
```



`scipy.special.`**`ncfdtridfd`**(*p*, *f*, *dfn*, *nc*) **= <ufunc 'ncfdtridfd'>**

Calculate degrees of freedom (denominator) for the noncentral F-distribution.

See *ncfdtr* for more details.

Notes

The value of the cumulative noncentral F distribution is not necessarily monotone in either degrees of freedom. There thus may be two values that provide a given CDF value. This routine assumes monotonicity and will find an arbitrary one of the two values.

`scipy.special.`**`ncfdtridfn`**(*p*, *f*, *dfd*, *nc*) **= <ufunc 'ncfdtridfn'>**

Calculate degrees of freedom (numerator) for the noncentral F-distribution.

See *ncfdtr* for more details.

Notes

The value of the cumulative noncentral F distribution is not necessarily monotone in either degrees of freedom. There thus may be two values that provide a given CDF value. This routine assumes monotonicity and will find an arbitrary one of the two values.

`scipy.special.`**`ncfdtri`**(*p*, *dfn*, *dfd*, *nc*) **= <ufunc 'ncfdtri'>**

Inverse cumulative distribution function of the non-central F distribution.

See *ncfdtr* for more details.

`scipy.special.`**`ncfdtrinc`**(*p*, *f*, *dfn*, *dfd*) **= <ufunc 'ncfdtrinc'>**

Calculate non-centrality parameter for non-central F distribution.

See *ncfdtr* for more details.

`scipy.special.`**`nctdtr`**(*df*, *nc*, *t*) **= <ufunc 'nctdtr'>**

Cumulative distribution function of the non-central *t* distribution.

Parameters

df : array_like
: Degrees of freedom of the distribution. Should be in range (0, inf).

nc : array_like
: Noncentrality parameter. Should be in range (-1e6, 1e6).

t : array_like
: Quantiles, i.e. the upper limit of integration.

Returns **cdf** : float or ndarray
The calculated CDF. If all inputs are scalar, the return will be a float. Otherwise it will be an array.

See also:

nctdtrit Inverse CDF (iCDF) of the non-central t distribution.
nctdtridf Calculate degrees of freedom, given CDF and iCDF values.
nctdtrinc Calculate non-centrality parameter, given CDF iCDF values.

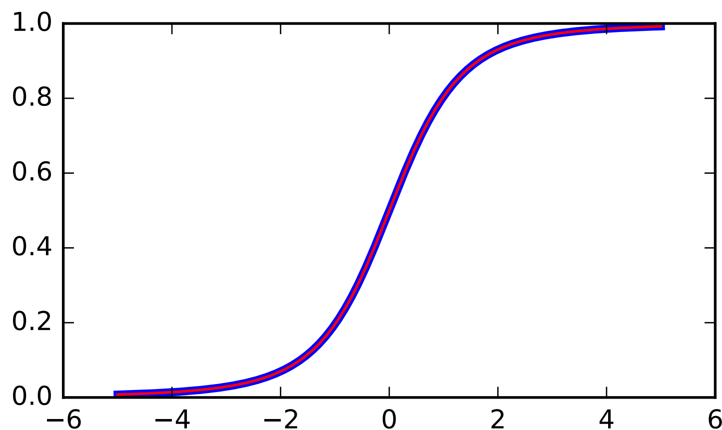
Examples

```
>>> from scipy import special
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

Plot the CDF of the non-central t distribution, for nc=0. Compare with the t-distribution from scipy.stats:

```
>>> x = np.linspace(-5, 5, num=500)
>>> df = 3
>>> nct_stats = stats.t.cdf(x, df)
>>> nct_special = special.nctdtr(df, 0, x)
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, nct_stats, 'b-', lw=3)
>>> ax.plot(x, nct_special, 'r-')
>>> plt.show()
```



scipy.special.**nctdtridf**(*p*, *nc*, *t*) = <ufunc 'nctdtridf'>
Calculate degrees of freedom for non-central t distribution.

See *nctdtr* for more details.

Parameters **p** : array_like
CDF values, in range (0, 1].
nc : array_like
Noncentrality parameter. Should be in range (-1e6, 1e6).
t : array_like

Quantiles, i.e. the upper limit of integration.

`scipy.special.`**`nctdtrit`**`(`*df*, *nc*, *p*`)` **= <ufunc 'nctdtrit'>**

Inverse cumulative distribution function of the non-central t distribution.

See *nctdtr* for more details.

Parameters

df : array_like
: Degrees of freedom of the distribution. Should be in range (0, inf).

nc : array_like
: Noncentrality parameter. Should be in range (-1e6, 1e6).

p : array_like
: CDF values, in range (0, 1].

`scipy.special.`**`nctdtrinc`**`(`*df*, *p*, *t*`)` **= <ufunc 'nctdtrinc'>**

Calculate non-centrality parameter for non-central t distribution.

See *nctdtr* for more details.

Parameters

df : array_like
: Degrees of freedom of the distribution. Should be in range (0, inf).

p : array_like
: CDF values, in range (0, 1].

t : array_like
: Quantiles, i.e. the upper limit of integration.

`scipy.special.`**`nrdtrimn`**`(`*p*, *x*, *std*`)` **= <ufunc 'nrdtrimn'>**

Calculate mean of normal distribution given other params.

Parameters

p : array_like
: CDF values, in range (0, 1].

x : array_like
: Quantiles, i.e. the upper limit of integration.

std : array_like
: Standard deviation.

Returns

mn : float or ndarray
: The mean of the normal distribution.

See also:

nrdtrimn, *ndtr*

`scipy.special.`**`nrdtrisd`**`(`*p*, *x*, *mn*`)` **= <ufunc 'nrdtrisd'>**

Calculate standard deviation of normal distribution given other params.

Parameters

p : array_like
: CDF values, in range (0, 1].

x : array_like
: Quantiles, i.e. the upper limit of integration.

mn : float or ndarray
: The mean of the normal distribution.

Returns

std : array_like
: Standard deviation.

See also:

`nrdtristd`, *ndtr*

`scipy.special.`**`pdtr`**`(`*k*, *m*`)` **= <ufunc 'pdtr'>**

Poisson cumulative distribution function

Returns the sum of the first *k* terms of the Poisson distribution: sum(exp(-m) * m**j / j!, j=0..k) = gammaincc(k+1, m). Arguments must both be positive and *k* an integer.

`scipy.special.pdtrc(k, m) = <ufunc 'pdtrc'>`
Poisson survival function

Returns the sum of the terms from k+1 to infinity of the Poisson distribution: sum(exp(-m) * m**j / j!, j=k+1..inf) = gammainc(k+1, m). Arguments must both be positive and *k* an integer.

`scipy.special.pdtri(k, y) = <ufunc 'pdtri'>`
Inverse to *pdtr* vs m

Returns the Poisson variable *m* such that the sum from 0 to *k* of the Poisson density is equal to the given probability *y*: calculated by gammaincinv(k+1, y). *k* must be a nonnegative integer and *y* between 0 and 1.

`scipy.special.pdtrik(p, m) = <ufunc 'pdtrik'>`
Inverse to *pdtr* vs k

Returns the quantile k such that `pdtr(k, m) = p`

`scipy.special.stdtr(df, t) = <ufunc 'stdtr'>`
Student t distribution cumulative density function

Returns the integral from minus infinity to t of the Student t distribution with df > 0 degrees of freedom:

```
gamma((df+1)/2)/(sqrt(df*pi)*gamma(df/2)) *
integral((1+x**2/df)**(-df/2-1/2), x=-inf..t)
```

`scipy.special.stdtridf(p, t) = <ufunc 'stdtridf'>`
Inverse of *stdtr* vs df

Returns the argument df such that stdtr(df, t) is equal to *p*.

`scipy.special.stdtrit(df, p) = <ufunc 'stdtrit'>`
Inverse of *stdtr* vs *t*

Returns the argument *t* such that stdtr(df, t) is equal to *p*.

`scipy.special.chdtr(v, x) = <ufunc 'chdtr'>`
Chi square cumulative distribution function

Returns the area under the left hand tail (from 0 to *x*) of the Chi square probability density function with *v* degrees of freedom:

```
1/(2**(v/2) * gamma(v/2)) * integral(t**(v/2-1) * exp(-t/2), t=0..x)
```

`scipy.special.chdtrc(v, x) = <ufunc 'chdtrc'>`
Chi square survival function

Returns the area under the right hand tail (from *x* to infinity) of the Chi square probability density function with *v* degrees of freedom:

```
1/(2**(v/2) * gamma(v/2)) * integral(t**(v/2-1) * exp(-t/2), t=x..inf)
```

`scipy.special.chdtri(v, p) = <ufunc 'chdtri'>`
Inverse to *chdtrc*

Returns the argument x such that `chdtrc(v, x) == p`.

`scipy.special.chdtriv(p, x) = <ufunc 'chdtriv'>`
Inverse to *chdtr* vs *v*

Returns the argument v such that `chdtr(v, x) == p`.

`scipy.special.ndtr(x) = <ufunc 'ndtr'>`
Gaussian cumulative distribution function.

Returns the area under the standard Gaussian probability density function, integrated from minus infinity to *x*

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp(-t^2/2) dt$$

Parameters **x** : array_like, real or complex

Argument

Returns ndarray

The value of the normal CDF evaluated at *x*

See also:

erf, *erfc*, *scipy.stats.norm*, *log_ndtr*

`scipy.special.`**`log_ndtr`**`(`*x*`)` **= <ufunc 'log_ndtr'>**

Logarithm of Gaussian cumulative distribution function.

Returns the log of the area under the standard Gaussian probability density function, integrated from minus infinity to *x*:

```
log(1/sqrt(2*pi) * integral(exp(-t**2 / 2), t=-inf..x))
```

Parameters **x** : array_like, real or complex

Argument

Returns ndarray

The value of the log of the normal CDF evaluated at *x*

See also:

erf, *erfc*, *scipy.stats.norm*, *ndtr*

`scipy.special.`**`ndtri`**`(`*y*`)` **= <ufunc 'ndtri'>**

Inverse of *ndtr* vs x

Returns the argument x for which the area under the Gaussian probability density function (integrated from minus infinity to *x*) is equal to y.

`scipy.special.`**`chndtr`**`(`*x, df, nc*`)` **= <ufunc 'chndtr'>**

Non-central chi square cumulative distribution function

`scipy.special.`**`chndtridf`**`(`*x, p, nc*`)` **= <ufunc 'chndtridf'>**

Inverse to *chndtr* vs *df*

`scipy.special.`**`chndtrinc`**`(`*x, df, p*`)` **= <ufunc 'chndtrinc'>**

Inverse to *chndtr* vs *nc*

`scipy.special.`**`chndtrix`**`(`*p, df, nc*`)` **= <ufunc 'chndtrix'>**

Inverse to *chndtr* vs *x*

`scipy.special.`**`smirnov`**`(`*n, e*`)` **= <ufunc 'smirnov'>**

Kolmogorov-Smirnov complementary cumulative distribution function

Returns the exact Kolmogorov-Smirnov complementary cumulative distribution function (Dn+ or Dn-) for a one-sided test of equality between an empirical and a theoretical distribution. It is equal to the probability that the maximum difference between a theoretical distribution and an empirical one based on *n* samples is greater than e.

`scipy.special.`**`smirnovi`**`(`*n, y*`)` **= <ufunc 'smirnovi'>**

Inverse to *smirnov*

Returns `e` such that `smirnov(n, e) = y`.

`scipy.special.kolmogorov(y) = <ufunc 'kolmogorov'>`

Complementary cumulative distribution function of Kolmogorov distribution

Returns the complementary cumulative distribution function of Kolmogorov's limiting distribution (Kn* for large n) of a two-sided test for equality between an empirical and a theoretical distribution. It is equal to the (limit as n->infinity of the) probability that sqrt(n) * max absolute deviation > y.

`scipy.special.kolmogi(p) = <ufunc 'kolmogi'>`

Inverse function to kolmogorov

Returns y such that `kolmogorov(y) == p`.

`scipy.special.tklmbda(x, lmbda) = <ufunc 'tklmbda'>`

Tukey-Lambda cumulative distribution function

`scipy.special.logit(x) = <ufunc 'logit'>`

Logit ufunc for ndarrays.

The logit function is defined as logit(p) = log(p/(1-p)). Note that logit(0) = -inf, logit(1) = inf, and logit(p) for p<0 or p>1 yields nan.

Parameters **x** : ndarray

The ndarray to apply logit to element-wise.

Returns **out** : ndarray

An ndarray of the same shape as x. Its entries are logit of the corresponding entry of x.

Notes

As a ufunc logit takes a number of optional keyword arguments. For more information see ufuncs

New in version 0.10.0.

`scipy.special.expit(x) = <ufunc 'expit'>`

Expit ufunc for ndarrays.

The expit function, also known as the logistic function, is defined as expit(x) = 1/(1+exp(-x)). It is the inverse of the logit function.

Parameters **x** : ndarray

The ndarray to apply expit to element-wise.

Returns **out** : ndarray

An ndarray of the same shape as x. Its entries are expit of the corresponding entry of x.

Notes

As a ufunc expit takes a number of optional keyword arguments. For more information see ufuncs

New in version 0.10.0.

`scipy.special.boxcox(x, lmbda) = <ufunc 'boxcox'>`

Compute the Box-Cox transformation.

The Box-Cox transformation is:

```
y = (x**lmbda - 1) / lmbda  if lmbda != 0
    log(x)                  if lmbda == 0
```

Returns *nan* if `x < 0`. Returns *-inf* if `x == 0` and `lmbda < 0`.

Parameters **x** : array_like

Data to be transformed.

lmbda : array_like

Returns **y** : array

Power parameter of the Box-Cox transform.

Transformed data.

Notes

New in version 0.14.0.

Examples

```
>>> from scipy.special import boxcox
>>> boxcox([1, 4, 10], 2.5)
array([   0.        ,   12.4       ,  126.09110641])
>>> boxcox(2, [0, 1, 2])
array([ 0.69314718,  1.        ,  1.5       ])
```

`scipy.special.`**`boxcox1p`**(*x*, *lmbda*) **= <ufunc 'boxcox1p'>**

Compute the Box-Cox transformation of $1 + x$.

The Box-Cox transformation computed by *boxcox1p* is:

```
y = ((1+x)**lmbda - 1) / lmbda  if lmbda != 0
    log(1+x)                    if lmbda == 0
```

Returns *nan* if `x < -1`. Returns *-inf* if `x == -1` and `lmbda < 0`.

Parameters **x** : array_like

Data to be transformed.

lmbda : array_like

Power parameter of the Box-Cox transform.

Returns **y** : array

Transformed data.

Notes

New in version 0.14.0.

Examples

```
>>> from scipy.special import boxcox1p
>>> boxcox1p(1e-4, [0, 0.5, 1])
array([  9.99950003e-05,   9.99975001e-05,   1.00000000e-04])
>>> boxcox1p([0.01, 0.1], 0.25)
array([ 0.00996272,  0.09645476])
```

`scipy.special.`**`inv_boxcox`**(*y*, *lmbda*) **= <ufunc 'inv_boxcox'>**

Compute the inverse of the Box-Cox transformation.

Find `x` such that:

```
y = (x**lmbda - 1) / lmbda  if lmbda != 0
    log(x)                  if lmbda == 0
```

Parameters **y** : array_like

Data to be transformed.

lmbda : array_like

Power parameter of the Box-Cox transform.

Returns **x** : array

Transformed data.

Notes

New in version 0.16.0.

Examples

```
>>> from scipy.special import boxcox, inv_boxcox
>>> y = boxcox([1, 4, 10], 2.5)
>>> inv_boxcox(y, 2.5)
array([1., 4., 10.])
```

`scipy.special.`**`inv_boxcox1p`**(*y*, *lmbda*) **= <ufunc 'inv_boxcox1p'>**

Compute the inverse of the Box-Cox transformation.

Find x such that:

```
y = ((1+x)**lmbda - 1) / lmbda  if lmbda != 0
    log(1+x)                    if lmbda == 0
```

Parameters

y : array_like
: Data to be transformed.

lmbda : array_like
: Power parameter of the Box-Cox transform.

Returns

x : array
: Transformed data.

Notes

New in version 0.16.0.

Examples

```
>>> from scipy.special import boxcox1p, inv_boxcox1p
>>> y = boxcox1p([1, 4, 10], 2.5)
>>> inv_boxcox1p(y, 2.5)
array([1., 4., 10.])
```

Information Theory Functions

entr(x)	Elementwise function for computing entropy.
rel_entr(x, y)	Elementwise function for computing relative entropy.
kl_div(x, y)	Elementwise function for computing Kullback-Leibler divergence.
huber(delta, r)	Huber loss function.
pseudo_huber(delta, r)	Pseudo-Huber loss function.

`scipy.special.`**`entr`**(*x*) **= <ufunc 'entr'>**

Elementwise function for computing entropy.

$$\text{entr}(x) = \begin{cases} -x\log(x) & x > 0 \\ 0 & x = 0 \\ -\infty & \text{otherwise} \end{cases}$$

Parameters

x : ndarray

Returns **res** : ndarray

Input array.

The value of the elementwise entropy function at the given points *x*.

See also:

kl_div, *rel_entr*

Notes

This function is concave.

New in version 0.15.0.

`scipy.special.rel_entr(x, y)` **= <ufunc 'rel_entr'>**

Elementwise function for computing relative entropy.

$$\text{rel_entr}(x, y) = \begin{cases} x \log(x/y) & x > 0, y > 0 \\ 0 & x = 0, y \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

Parameters **x** : ndarray

First input array.

y : ndarray

Second input array.

Returns **res** : ndarray

Output array.

See also:

entr, *kl_div*

Notes

This function is jointly convex in x and y.

New in version 0.15.0.

`scipy.special.kl_div(x, y)` **= <ufunc 'kl_div'>**

Elementwise function for computing Kullback-Leibler divergence.

$$\text{kl_div}(x, y) = \begin{cases} x \log(x/y) - x + y & x > 0, y > 0 \\ y & x = 0, y \geq 0 \\ \infty & \text{otherwise} \end{cases}$$

Parameters **x** : ndarray

First input array.

y : ndarray

Second input array.

Returns **res** : ndarray

Output array.

See also:

entr, *rel_entr*

Notes

This function is non-negative and is jointly convex in *x* and *y*.

New in version 0.15.0.

`scipy.special.huber(delta, r) = <ufunc 'huber'>`

Huber loss function.

$$\text{huber}(\delta, r) = \begin{cases} \infty & \delta < 0 \\ \frac{1}{2}r^2 & 0 \le \delta, |r| \le \delta \\ \delta(|r| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Parameters **delta** : ndarray

Input array, indicating the quadratic vs. linear loss changepoint.

r : ndarray

Input array, possibly representing residuals.

Returns **res** : ndarray

The computed Huber loss function values.

Notes

This function is convex in r.

New in version 0.15.0.

`scipy.special.pseudo_huber(delta, r) = <ufunc 'pseudo_huber'>`

Pseudo-Huber loss function.

$$\text{pseudo_huber}(\delta, r) = \delta^2 \left(\sqrt{1 + \left(\frac{r}{\delta}\right)^2} - 1 \right)$$

Parameters **delta** : ndarray

Input array, indicating the soft quadratic vs. linear loss changepoint.

r : ndarray

Input array, possibly representing residuals.

Returns **res** : ndarray

The computed Pseudo-Huber loss function values.

Notes

This function is convex in r.

New in version 0.15.0.

Gamma and Related Functions

gamma(z)	Gamma function.
gammaln(x)	Logarithm of the absolute value of the Gamma function for real inputs.
loggamma(z[, out])	Principal branch of the logarithm of the Gamma function.
gammasgn(x)	Sign of the gamma function.
gammainc(a, x)	Regularized lower incomplete gamma function.
gammaincinv(a, y)	Inverse to *gammainc*
gammaincc(a, x)	Regularized upper incomplete gamma function.
gammainccinv(a, y)	Inverse to *gammaincc*
beta(a, b)	Beta function.
betaln(a, b)	Natural logarithm of absolute value of beta function.
betainc(a, b, x)	Incomplete beta integral.
betaincinv(a, b, y)	Inverse function to beta integral.
psi(z[, out])	The digamma function.

Continued on next page

Table 5.244 – continued from previous page

rgamma(z)	Gamma function inverted
polygamma(n, x)	Polygamma function n.
multigammaln(a, d)	Returns the log of multivariate gamma, also sometimes called the generalized gamma.
digamma(z[, out])	The digamma function.
poch(z, m)	Rising factorial (z)_m

scipy.special.**gamma**(*z*) = <ufunc 'gamma'>

Gamma function.

The gamma function is often referred to as the generalized factorial since `z*gamma(z) = gamma(z+1)` and `gamma(n+1) = n!` for natural number *n*.

scipy.special.**gammaln**(*x*)

Logarithm of the absolute value of the Gamma function for real inputs.

Parameters **x** : array-like
Values on the real line at which to compute `gammaln`

Returns **gammaln** : ndarray
Values of `gammaln` at x.

See also:

gammasgn sign of the gamma function
loggamma principal branch of the logarithm of the gamma function

Notes

When used in conjunction with *gammasgn*, this function is useful for working in logspace on the real axis without having to deal with complex numbers, via the relation `exp(gammaln(x)) = gammasgn(x)*gamma(x)`.

Note that *gammaln* currently accepts complex-valued inputs, but it is not the same function as for real-valued inputs, and the branch is not well-defined — using *gammaln* with complex is deprecated and will be disallowed in future Scipy versions.

For complex-valued log-gamma, use *loggamma* instead of *gammaln*.

scipy.special.**loggamma**(*z*, *out=None*) = <ufunc 'loggamma'>

Principal branch of the logarithm of the Gamma function.

Defined to be $\log(\Gamma(x))$ for $x > 0$ and extended to the complex plane by analytic continuation. The function has a single branch cut on the negative real axis.

New in version 0.18.0.

Parameters **z** : array-like
Values in the complex plain at which to compute `loggamma`
out : ndarray, optional
Output array for computed values of `loggamma`

Returns **loggamma** : ndarray
Values of `loggamma` at z.

See also:

gammaln logarithm of the absolute value of the Gamma function
gammasgn sign of the gamma function

Notes

It is not generally true that $\log \Gamma(z) = \log(\Gamma(z))$, though the real parts of the functions do agree. The benefit of not defining `loggamma` as $\log(\Gamma(z))$ is that the latter function has a complicated branch cut structure whereas `loggamma` is analytic except for on the negative real axis.

The identities

$$\exp(\log \Gamma(z)) = \Gamma(z)$$
$$\log \Gamma(z+1) = \log(z) + \log \Gamma(z)$$

make `loggama` useful for working in complex logspace. However, `loggamma` necessarily returns complex outputs for real inputs, so if you want to work only with real numbers use *gammaln*. On the real line the two functions are related by `exp(loggamma(x)) = gammasgn(x)*exp(gammaln(x))`, though in practice rounding errors will introduce small spurious imaginary components in `exp(loggamma(x))`.

The implementation here is based on *[hare1997]*.

References

[hare1997]

`scipy.special.`**gammasgn**(*x*) **= <ufunc 'gammasgn'>**

Sign of the gamma function.

See also:

gammaln, *loggamma*

`scipy.special.`**gammainc**(*a*, *x*) **= <ufunc 'gammainc'>**

Regularized lower incomplete gamma function.

Defined as

$$\frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

for $a > 0$ and $x \geq 0$. The function satisfies the relation `gammainc(a, x) + gammaincc(a, x) = 1` where *gammaincc* is the regularized upper incomplete gamma function.

See also:

gammaincc regularized upper incomplete gamma function

gammaincinv
inverse to `gammainc` versus `x`

gammainccinv
inverse to `gammaincc` versus `x`

Notes

The implementation largely follows that of *[R421]*.

References

[R421]

`scipy.special.`**gammaincinv**(*a*, *y*) **= <ufunc 'gammaincinv'>**

Inverse to *gammainc*

Returns *x* such that `gammainc(a, x) = y`.

`scipy.special.`**gammaincc**(*a*, *x*) **= <ufunc 'gammaincc'>**

Regularized upper incomplete gamma function.

Defined as

$$\frac{1}{\Gamma(a)} \int_x^{\infty} t^{a-1} e^{-t} dt$$

for $a > 0$ and $x \geq 0$. The function satisfies the relation `gammainc(a, x) + gammaincc(a, x) = 1` where *gammainc* is the regularized lower incomplete gamma function.

See also:

gammainc regularized lower incomplete gamma function

gammaincinv
inverse to `gammainc` versus `x`

gammainccinv
inverse to `gammaincc` versus `x`

Notes

The implementation largely follows that of *[R422]*.

References

[R422]

`scipy.special.`**gammainccinv**(*a*, *y*) = **<ufunc 'gammainccinv'>**

Inverse to *gammaincc*

Returns *x* such that `gammaincc(a, x) == y`.

`scipy.special.`**beta**(*a*, *b*) = **<ufunc 'beta'>**

Beta function.

```
beta(a, b) =  gamma(a) * gamma(b) / gamma(a+b)
```

`scipy.special.`**betaln**(*a*, *b*) = **<ufunc 'betaln'>**

Natural logarithm of absolute value of beta function.

Computes `ln(abs(beta(a, b)))`.

`scipy.special.`**betainc**(*a*, *b*, *x*) = **<ufunc 'betainc'>**

Incomplete beta integral.

Compute the incomplete beta integral of the arguments, evaluated from zero to *x*:

```
gamma(a+b) / (gamma(a)*gamma(b)) * integral(t**(a-1) (1-t)**(b-1), t=0..x).
```

Notes

The incomplete beta is also sometimes defined without the terms in gamma, in which case the above definition is the so-called regularized incomplete beta. Under this definition, you can get the incomplete beta by multiplying the result of the scipy function by beta(a, b).

`scipy.special.`**betaincinv**(*a*, *b*, *y*) = **<ufunc 'betaincinv'>**

Inverse function to beta integral.

Compute *x* such that betainc(a, b, x) = y.

`scipy.special.`**psi**(*z*, *out=None*) = **<ufunc 'psi'>**

The digamma function.

The logarithmic derivative of the gamma function evaluated at `z`.

Parameters **z** : array_like

Real or complex argument.

out : ndarray, optional
Array for the computed values of `psi`.

Returns **digamma** : ndarray
Computed values of `psi`.

Notes

For large values not close to the negative real axis `psi` is computed using the asymptotic series (5.11.2) from *[R508]*. For small arguments not close to the negative real axis the recurrence relation (5.5.2) from *[R508]* is used until the argument is large enough to use the asymptotic series. For values close to the negative real axis the reflection formula (5.5.4) from *[R508]* is used first. Note that `psi` has a family of zeros on the negative real axis which occur between the poles at nonpositive integers. Around the zeros the reflection formula suffers from cancellation and the implementation loses precision. The sole positive zero and the first negative zero, however, are handled separately by precomputing series expansions using *[R509]*, so the function should maintain full accuracy around the origin.

References

[R508], *[R509]*

`scipy.special.`**`rgamma`**(*z*) = **<ufunc 'rgamma'>**

Gamma function inverted

Returns `1/gamma(x)`

`scipy.special.`**`polygamma`**(*n*, *x*)

Polygamma function n.

This is the nth derivative of the digamma (psi) function.

Parameters **n** : array_like of int
The order of the derivative of *psi*.

x : array_like
Where to evaluate the polygamma function.

Returns **polygamma** : ndarray
The result.

Examples

```
>>> from scipy import special
>>> x = [2, 3, 25.5]
>>> special.polygamma(1, x)
array([ 0.64493407,  0.39493407,  0.03999467])
>>> special.polygamma(0, x) == special.psi(x)
array([ True,  True,  True], dtype=bool)
```

`scipy.special.`**`multigammaln`**(*a*, *d*)

Returns the log of multivariate gamma, also sometimes called the generalized gamma.

Parameters **a** : ndarray
The multivariate gamma is computed for each item of *a*.

d : int
The dimension of the space of integration.

Returns **res** : ndarray
The values of the log multivariate gamma at the given points *a*.

Notes

The formal definition of the multivariate gamma of dimension d for a real *a* is

$$\Gamma_d(a) = \int_{A>0} e^{-tr(A)} |A|^{a-(d+1)/2} dA$$

with the condition $a > (d-1)/2$, and $A > 0$ being the set of all the positive definite matrices of dimension *d*. Note that *a* is a scalar: the integrand only is multivariate, the argument is not (the function is defined over a subset of the real set).

This can be proven to be equal to the much friendlier equation

$$\Gamma_d(a) = \pi^{d(d-1)/4} \prod_{i=1}^{d} \Gamma(a - (i-1)/2).$$

References

R. J. Muirhead, Aspects of multivariate statistical theory (Wiley Series in probability and mathematical statistics).

`scipy.special.`**`digamma`**(*z*, *out=None*) **= <ufunc 'psi'>**

The digamma function.

The logarithmic derivative of the gamma function evaluated at `z`.

Parameters
- **z** : array_like
  Real or complex argument.
- **out** : ndarray, optional
  Array for the computed values of `psi`.

Returns
- **digamma** : ndarray
  Computed values of `psi`.

Notes

For large values not close to the negative real axis `psi` is computed using the asymptotic series (5.11.2) from *[R396]*. For small arguments not close to the negative real axis the recurrence relation (5.5.2) from *[R396]* is used until the argument is large enough to use the asymptotic series. For values close to the negative real axis the reflection formula (5.5.4) from *[R396]* is used first. Note that `psi` has a family of zeros on the negative real axis which occur between the poles at nonpositive integers. Around the zeros the reflection formula suffers from cancellation and the implementation loses precision. The sole positive zero and the first negative zero, however, are handled separately by precomputing series expansions using *[R397]*, so the function should maintain full accuracy around the origin.

References

[R396], *[R397]*

`scipy.special.`**`poch`**(*z*, *m*) **= <ufunc 'poch'>**

Rising factorial (z)_m

The Pochhammer symbol (rising factorial), is defined as:

```
(z)_m = gamma(z + m) / gamma(z)
```

For positive integer *m* it reads:

```
(z)_m = z * (z + 1) * ... * (z + m - 1)
```

Error Function and Fresnel Integrals

erf(z)	Returns the error function of complex argument.
erfc(x)	Complementary error function, `1 - erf(x)`.
erfcx(x)	Scaled complementary error function, `exp(x**2) * erfc(x)`.
erfi(z)	Imaginary error function, `-i erf(i z)`.
erfinv(y)	Inverse function for erf.
erfcinv(y)	Inverse function for erfc.
wofz(z)	Faddeeva function
dawsn(x)	Dawson's integral.
fresnel(z)	Fresnel sin and cos integrals
fresnel_zeros(nt)	Compute nt complex zeros of sine and cosine Fresnel integrals S(z) and C(z).
modfresnelp(x)	Modified Fresnel positive integrals
modfresnelm(x)	Modified Fresnel negative integrals

`scipy.special.`**`erf`**`(z)` **= <ufunc 'erf'>**

Returns the error function of complex argument.

It is defined as `2/sqrt(pi)*integral(exp(-t**2), t=0..z)`.

Parameters **x** : ndarray

Input array.

Returns **res** : ndarray

The values of the error function at the given points *x*.

See also:

erfc, *erfinv*, *erfcinv*, *wofz*, *erfcx*, *erfi*

Notes

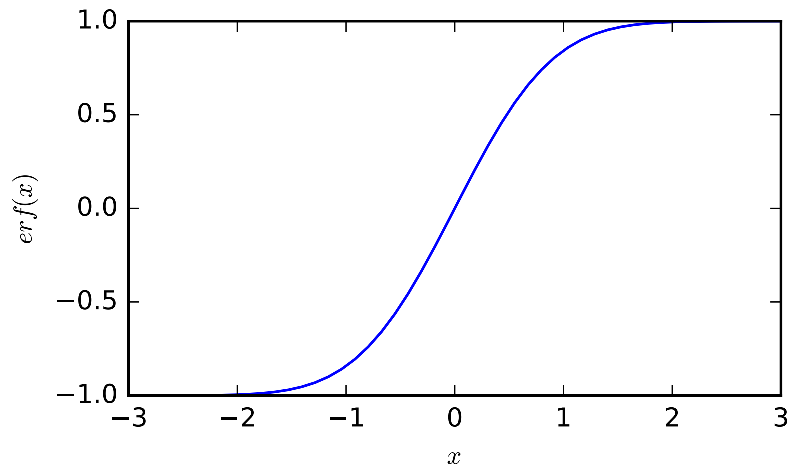
The cumulative of the unit normal distribution is given by `Phi(z) = 1/2[1 + erf(z/sqrt(2))]`.

References

[R407], *[R408]*, *[R409]*

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-3, 3)
>>> plt.plot(x, special.erf(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$erf(x)$')
>>> plt.show()
```



`scipy.special.erfc(x)` = <ufunc 'erfc'>

Complementary error function, `1 - erf(x)`.

See also:

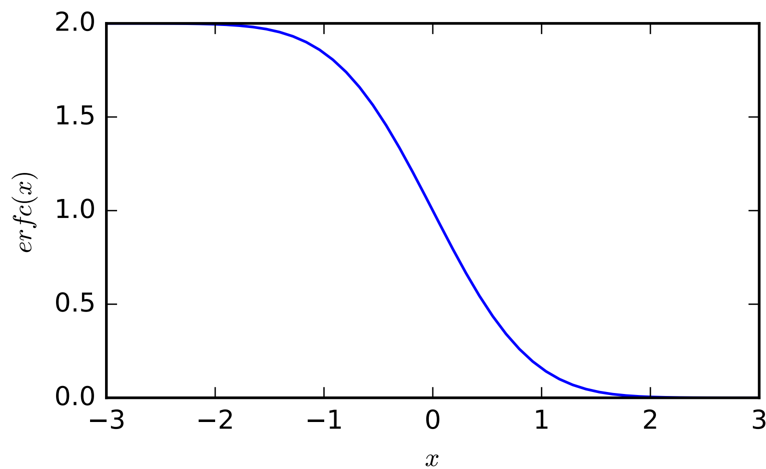
erf, *erfi*, *erfcx*, *dawsn*, *wofz*

References

[R411]

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-3, 3)
>>> plt.plot(x, special.erfc(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$erfc(x)$')
>>> plt.show()
```



`scipy.special.erfcx(x) = <ufunc 'erfcx'>`

Scaled complementary error function, `exp(x**2) * erfc(x)`.

See also:

erf, *erfc*, *erfi*, *dawsn*, *wofz*

Notes

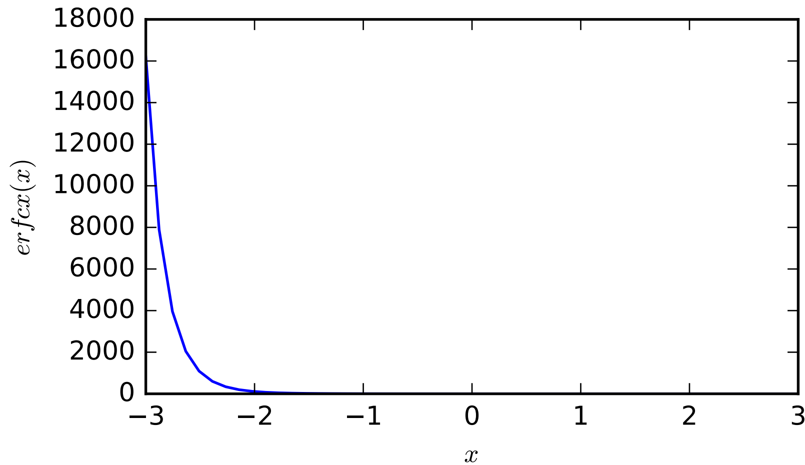
New in version 0.12.0.

References

[R412]

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-3, 3)
>>> plt.plot(x, special.erfcx(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$erfcx(x)$')
>>> plt.show()
```



`scipy.special.erfi(z) = <ufunc 'erfi'>`

Imaginary error function, `-i erf(i z)`.

See also:

erf, *erfc*, *erfcx*, *dawsn*, *wofz*

Notes

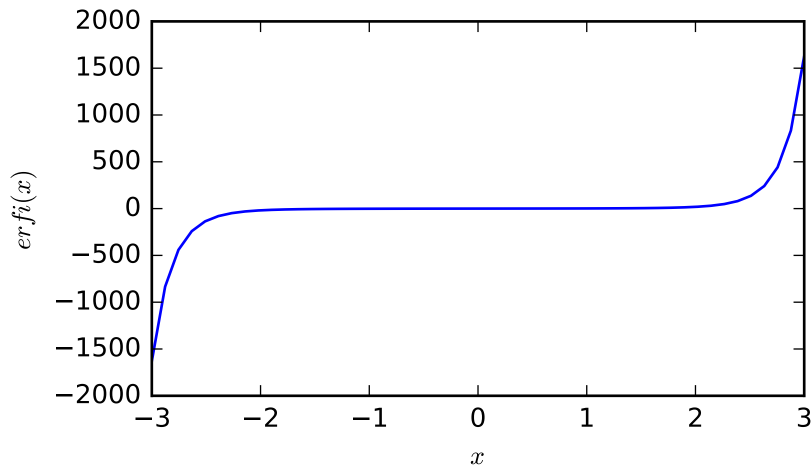
New in version 0.12.0.

References

[R413]

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-3, 3)
>>> plt.plot(x, special.erfi(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$erfi(x)$')
>>> plt.show()
```



scipy.special.**erfinv**(*y*)
Inverse function for erf.

scipy.special.**erfcinv**(*y*)
Inverse function for erfc.

scipy.special.**wofz**(*z*) = <ufunc 'wofz'>
Faddeeva function

Returns the value of the Faddeeva function for complex argument:

```
exp(-z**2) * erfc(-i*z)
```

See also:

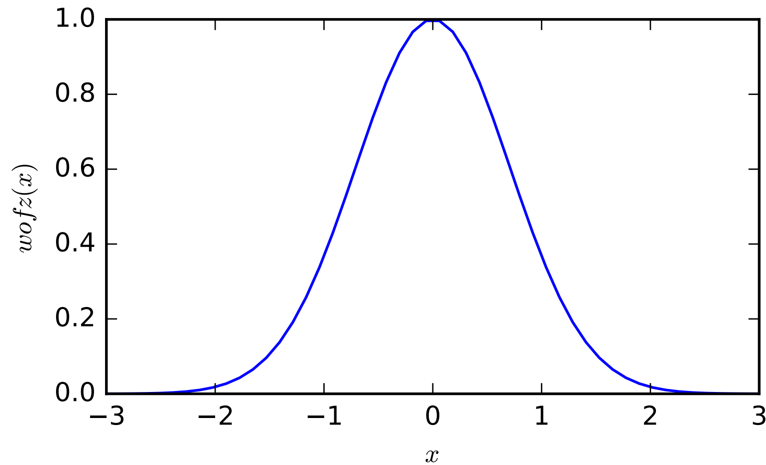
dawsn, *erf*, *erfc*, *erfcx*, *erfi*

References

[R533]

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-3, 3)
>>> plt.plot(x, special.wofz(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$wofz(x)$')
>>> plt.show()
```



scipy.special.**dawsn**(*x*) = <ufunc 'dawsn'>

Dawson's integral.

Computes:

```
exp(-x**2) * integral(exp(t**2), t=0..x).
```

See also:

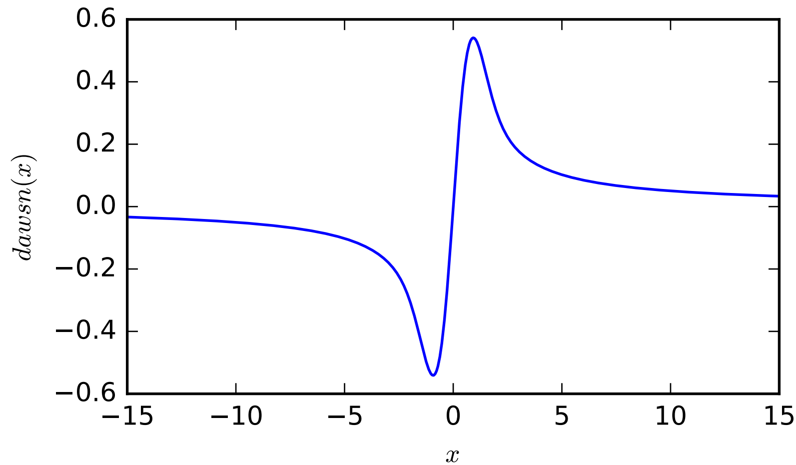
wofz, *erf*, *erfc*, *erfcx*, *erfi*

References

[R395]

Examples

```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-15, 15, num=1000)
>>> plt.plot(x, special.dawsn(x))
>>> plt.xlabel('$x$')
>>> plt.ylabel('$dawsn(x)$')
>>> plt.show()
```



scipy.special.**fresnel**(*z*) = <ufunc 'fresnel'>

Fresnel sin and cos integrals

Defined as:

```
ssa = integral(sin(pi/2 * t**2), t=0..z)
csa = integral(cos(pi/2 * t**2), t=0..z)
```

Parameters **z** : float or complex array_like

Argument

Returns ssa, csa

Fresnel sin and cos integral values

scipy.special.**fresnel_zeros**(*nt*)

Compute nt complex zeros of sine and cosine Fresnel integrals S(z) and C(z).

References

[R418]

scipy.special.**modfresnelp**(*x*) = <ufunc 'modfresnelp'>

Modified Fresnel positive integrals

Returns fp

Integral `F_+(x): integral(exp(1j*t*t), t=x..inf)`

kp

Integral `K_+(x): 1/sqrt(pi)*exp(-1j*(x*x+pi/4))*fp`

scipy.special.**modfresnelm**(*x*) = <ufunc 'modfresnelm'>

Modified Fresnel negative integrals

Returns fm

Integral `F_-(x): integral(exp(-1j*t*t), t=x..inf)`

km

Integral `K_-(x): 1/sqrt(pi)*exp(1j*(x*x+pi/4))*fp`

These are not universal functions:

Continued on next page

Table 5.246 – continued from previous page

erf_zeros(nt)	Compute nt complex zeros of error function erf(z).
fresnelc_zeros(nt)	Compute nt complex zeros of cosine Fresnel integral C(z).
fresnels_zeros(nt)	Compute nt complex zeros of sine Fresnel integral S(z).

`scipy.special.erf_zeros(nt)`
Compute nt complex zeros of error function erf(z).

References

[R410]

`scipy.special.fresnelc_zeros(nt)`
Compute nt complex zeros of cosine Fresnel integral C(z).

References

[R419]

`scipy.special.fresnels_zeros(nt)`
Compute nt complex zeros of sine Fresnel integral S(z).

References

[R420]

Legendre Functions

lpmv(m, v, x)	Associated Legendre function of integer order and real degree.
sph_harm(m, n, theta, phi)	Compute spherical harmonics.

`scipy.special.lpmv(m, v, x) = <ufunc 'lpmv'>`
Associated Legendre function of integer order and real degree.

Defined as

$$P_v^m = (-1)^m (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_v(x)$$

where

$$P_v = \sum_{k=0}^{\infty} \frac{(-v)_k (v+1)_k}{(k!)^2} \left(\frac{1-x}{2}\right)^k$$

is the Legendre function of the first kind. Here $(\cdot)_k$ is the Pochhammer symbol; see *poch*.

Parameters

m : array_like
: Order (int or float). If passed a float not equal to an integer the function returns NaN.

v : array_like
: Degree (float).

x : array_like
: Argument (float). Must have `|x| <= 1`.

Returns

pmv : ndarray
: Value of the associated Legendre function.

See also:

lpmn Compute the associated Legendre function for all orders `0, ..., m` and degrees `0, ..., n`.

clpmn Compute the associated Legendre function at complex arguments.

Notes

Note that this implementation includes the Condon-Shortley phase.

References

[R486]

`scipy.special.`**`sph_harm`**(*m*, *n*, *theta*, *phi*) **= <ufunc 'sph_harm'>**

Compute spherical harmonics.

The spherical harmonics are defined as

$$Y^m_n(\theta,\phi) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}} e^{im\theta} P^m_n(\cos(\phi))$$

where P_n^m are the associated Legendre functions; see *lpmv*.

Parameters

m : array_like
: Order of the harmonic (int); must have `|m| <= n`.

n : array_like
: Degree of the harmonic (int); must have `n >= 0`. This is often denoted by `l` (lower case L) in descriptions of spherical harmonics.

theta : array_like
: Azimuthal (longitudinal) coordinate; must be in `[0, 2*pi]`.

phi : array_like
: Polar (colatitudinal) coordinate; must be in `[0, pi]`.

Returns

y_mn : complex float
: The harmonic Y^m_n sampled at `theta` and `phi`.

Notes

There are different conventions for the meanings of the input arguments `theta` and `phi`. In SciPy `theta` is the azimuthal angle and `phi` is the polar angle. It is common to see the opposite convention, that is, `theta` as the polar angle and `phi` as the azimuthal angle.

Note that SciPy's spherical harmonics include the Condon-Shortley phase *[R521]* because it is part of *lpmv*.

With SciPy's conventions, the first several spherical harmonics are

$$Y_0^0(\theta, \phi) = \frac{1}{2} \sqrt{\frac{1}{\pi}}$$

$$Y_1^{-1}(\theta, \phi) = \frac{1}{2} \sqrt{\frac{3}{2\pi}} e^{-i\theta} \sin(\phi)$$

$$Y_1^0(\theta, \phi) = \frac{1}{2} \sqrt{\frac{3}{\pi}} \cos(\phi)$$

$$Y_1^1(\theta, \phi) = -\frac{1}{2} \sqrt{\frac{3}{2\pi}} e^{i\theta} \sin(\phi).$$

References

[R520], *[R521]*

These are not universal functions:

clpmn(m, n, z[, type])	Associated Legendre function of the first kind for complex arguments.
lpn(n, z)	Legendre function of the first kind.
lqn(n, z)	Legendre function of the second kind.
lpmn(m, n, z)	Sequence of associated Legendre functions of the first kind.
lqmn(m, n, z)	Sequence of associated Legendre functions of the second kind.

`scipy.special.`**`clpmn`**(*m*, *n*, *z*, *type=3*)

Associated Legendre function of the first kind for complex arguments.

Computes the associated Legendre function of the first kind of order m and degree n, `Pmn(z)` = $P_n^m(z)$, and its derivative, `Pmn'(z)`. Returns two arrays of size `(m+1, n+1)` containing `Pmn(z)` and `Pmn'(z)` for all orders from `0..m` and degrees from `0..n`.

Parameters

m : int
: `|m| <= n`; the order of the Legendre function.

n : int
: where `n >= 0`; the degree of the Legendre function. Often called `l` (lower case L) in descriptions of the associated Legendre function

z : float or complex
: Input value.

type : int, optional
: takes values 2 or 3 2: cut on the real axis `|x| > 1` 3: cut on the real axis `-1 < x < 1` (default)

Returns

Pmn_z : (m+1, n+1) array
: Values for all orders `0..m` and degrees `0..n`

Pmn_d_z : (m+1, n+1) array
: Derivatives for all orders `0..m` and degrees `0..n`

See also:

`lpmn` associated Legendre functions of the first kind for real z

Notes

By default, i.e. for `type=3`, phase conventions are chosen according to *[R392]* such that the function is analytic. The cut lies on the interval (-1, 1). Approaching the cut from above or below in general yields a phase factor with respect to Ferrer's function of the first kind (cf. *lpmn*).

For `type=2` a cut at `|x| > 1` is chosen. Approaching the real values on the interval (-1, 1) in the complex plane yields Ferrer's function of the first kind.

References

[R392], *[R393]*

`scipy.special.`**`lpn`**(*n*, *z*)

Legendre function of the first kind.

Compute sequence of Legendre functions of the first kind (polynomials), Pn(z) and derivatives for all degrees from 0 to n (inclusive).

See also special.legendre for polynomial class.

References

[R487]

`scipy.special.lqn(n, z)`

Legendre function of the second kind.

Compute sequence of Legendre functions of the second kind, Qn(z) and derivatives for all degrees from 0 to n (inclusive).

References

[R489]

`scipy.special.lpmn(m, n, z)`

Sequence of associated Legendre functions of the first kind.

Computes the associated Legendre function of the first kind of order m and degree n, `Pmn(z)` = $P_n^m(z)$, and its derivative, `Pmn'(z)`. Returns two arrays of size `(m+1, n+1)` containing `Pmn(z)` and `Pmn'(z)` for all orders from `0..m` and degrees from `0..n`.

This function takes a real argument `z`. For complex arguments `z` use clpmn instead.

Parameters

m : int
: `|m| <= n`; the order of the Legendre function.

n : int
: where `n >= 0`; the degree of the Legendre function. Often called `l` (lower case L) in descriptions of the associated Legendre function

z : float
: Input value.

Returns

Pmn_z : (m+1, n+1) array
: Values for all orders 0..m and degrees 0..n

Pmn_d_z : (m+1, n+1) array
: Derivatives for all orders 0..m and degrees 0..n

See also:

`clpmn` associated Legendre functions of the first kind for complex z

Notes

In the interval (-1, 1), Ferrer's function of the first kind is returned. The phase convention used for the intervals (1, inf) and (-inf, -1) is such that the result is always real.

References

[R484], [R485]

`scipy.special.lqmn(m, n, z)`

Sequence of associated Legendre functions of the second kind.

Computes the associated Legendre function of the second kind of order m and degree n, `Qmn(z)` = $Q_n^m(z)$, and its derivative, `Qmn'(z)`. Returns two arrays of size `(m+1, n+1)` containing `Qmn(z)` and `Qmn'(z)` for all orders from `0..m` and degrees from `0..n`.

Parameters

m : int
: `|m| <= n`; the order of the Legendre function.

n : int
: where `n >= 0`; the degree of the Legendre function. Often called `l` (lower case L) in descriptions of the associated Legendre function

z : complex
: Input value.

Returns

Qmn_z : (m+1, n+1) array

Values for all orders 0..m and degrees 0..n

Qmn_d_z : (m+1, n+1) array

Derivatives for all orders 0..m and degrees 0..n

References

[R488]

Ellipsoidal Harmonics

ellip_harm(h2, k2, n, p, s[, signm, signn])	Ellipsoidal harmonic functions E^p_n(l)
ellip_harm_2(h2, k2, n, p, s)	Ellipsoidal harmonic functions F^p_n(l)
ellip_normal(h2, k2, n, p)	Ellipsoidal harmonic normalization constants gamma^p_n

scipy.special.**ellip_harm**(*h2*, *k2*, *n*, *p*, *s*, *signm=1*, *signn=1*)

Ellipsoidal harmonic functions E^p_n(l)

These are also known as Lame functions of the first kind, and are solutions to the Lame equation:

$$(s^2 - h^2)(s^2 - k^2)E''(s) + s(2s^2 - h^2 - k^2)E'(s) + (a - qs^2)E(s) = 0$$

where $q = (n + 1)n$ and a is the eigenvalue (not returned) corresponding to the solutions.

Parameters

h2 : float

h**2

k2 : float

k**2; should be larger than h**2

n : int

Degree

s : float

Coordinate

p : int

Order, can range between [1,2n+1]

signm : {1, -1}, optional

Sign of prefactor of functions. Can be +/-1. See Notes.

signn : {1, -1}, optional

Sign of prefactor of functions. Can be +/-1. See Notes.

Returns

E : float

the harmonic $E^p_n(s)$

See also:

ellip_harm_2, *ellip_normal*

Notes

The geometric intepretation of the ellipsoidal functions is explained in *[R399]*, *[R400]*, *[R401]*. The *signm* and *signn* arguments control the sign of prefactors for functions according to their type:

```
K : +1
L : signm
M : signn
N : signm*signn
```

New in version 0.15.0.

References

[R398], [R399], [R400], [R401]

Examples

```
>>> from scipy.special import ellip_harm
>>> w = ellip_harm(5,8,1,1,2.5)
>>> w
2.5
```

Check that the functions indeed are solutions to the Lame equation:

```
>>> from scipy.interpolate import UnivariateSpline
>>> def eigenvalue(f, df, ddf):
...     r = ((s**2 - h**2)*(s**2 - k**2)*ddf + s*(2*s**2 - h**2 - k**2)*df -
↪n*(n+1)*s**2*f)/f
...     return -r.mean(), r.std()
>>> s = np.linspace(0.1, 10, 200)
>>> k, h, n, p = 8.0, 2.2, 3, 2
>>> E = ellip_harm(h**2, k**2, n, p, s)
>>> E_spl = UnivariateSpline(s, E)
>>> a, a_err = eigenvalue(E_spl(s), E_spl(s,1), E_spl(s,2))
>>> a, a_err
(583.44366156701483, 6.4580890640310646e-11)
```

`scipy.special.`**`ellip_harm_2`**(*h2*, *k2*, *n*, *p*, *s*)

Ellipsoidal harmonic functions F^p_n(l)

These are also known as Lame functions of the second kind, and are solutions to the Lame equation:

$$(s^2 - h^2)(s^2 - k^2)F''(s) + s(2s^2 - h^2 - k^2)F'(s) + (a - qs^2)F(s) = 0$$

where $q = (n + 1)n$ and a is the eigenvalue (not returned) corresponding to the solutions.

Parameters

h2 : float

`h**2`

k2 : float

`k**2`; should be larger than `h**2`

n : int

Degree.

p : int

Order, can range between [1,2n+1].

s : float

Coordinate

Returns

F : float

The harmonic $F_n^p(s)$

See also:

`ellip_harm`, *`ellip_normal`*

Notes

Lame functions of the second kind are related to the functions of the first kind:

$$F_n^p(s) = (2n + 1)E_n^p(s) \int_0^{1/s} \frac{du}{(E_n^p(1/u))^2 \sqrt{(1 - u^2k^2)(1 - u^2h^2)}}$$

New in version 0.15.0.

Examples

```
>>> from scipy.special import ellip_harm_2
>>> w = ellip_harm_2(5,8,2,1,10)
>>> w
0.00108056853382
```

`scipy.special.ellip_normal(h2, k2, n, p)`

Ellipsoidal harmonic normalization constants gamma^p_n

The normalization constant is defined as

$$\gamma^p_n = 8 \int_0^h dx \int_h^k dy \frac{(y^2 - x^2)(E^p_n(y)E^p_n(x))^2}{\sqrt{((k^2 - y^2)(y^2 - h^2)(h^2 - x^2)(k^2 - x^2)}}$$

Parameters

h2 : float

`h**2`

k2 : float

`k**2`; should be larger than `h**2`

n : int

Degree.

p : int

Order, can range between [1,2n+1].

Returns

gamma : float

The normalization constant γ^p_n

See also:

ellip_harm, *ellip_harm_2*

Notes

New in version 0.15.0.

Examples

```
>>> from scipy.special import ellip_normal
>>> w = ellip_normal(5,8,3,7)
>>> w
1723.38796997
```

Orthogonal polynomials

The following functions evaluate values of orthogonal polynomials:

assoc_laguerre(x, n[, k])	Compute the generalized (associated) Laguerre polynomial of degree n and order k.
eval_legendre(n, x[, out])	Evaluate Legendre polynomial at a point.
eval_chebyt(n, x[, out])	Evaluate Chebyshev polynomial of the first kind at a point.
eval_chebyu(n, x[, out])	Evaluate Chebyshev polynomial of the second kind at a point.
eval_chebyc(n, x[, out])	Evaluate Chebyshev polynomial of the first kind on [-2, 2] at a point.
eval_chebys(n, x[, out])	Evaluate Chebyshev polynomial of the second kind on [-2, 2] at a point.
eval_jacobi(n, alpha, beta, x[, out])	Evaluate Jacobi polynomial at a point.

Continued on next page

Table 5.250 – continued from previous page

eval_laguerre(n, x[, out])	Evaluate Laguerre polynomial at a point.
eval_genlaguerre(n, alpha, x[, out])	Evaluate generalized Laguerre polynomial at a point.
eval_hermite(n, x[, out])	Evaluate physicist's Hermite polynomial at a point.
eval_hermitenorm(n, x[, out])	Evaluate probabilist's (normalized) Hermite polynomial at a point.
eval_gegenbauer(n, alpha, x[, out])	Evaluate Gegenbauer polynomial at a point.
eval_sh_legendre(n, x[, out])	Evaluate shifted Legendre polynomial at a point.
eval_sh_chebyt(n, x[, out])	Evaluate shifted Chebyshev polynomial of the first kind at a point.
eval_sh_chebyu(n, x[, out])	Evaluate shifted Chebyshev polynomial of the second kind at a point.
eval_sh_jacobi(n, p, q, x[, out])	Evaluate shifted Jacobi polynomial at a point.

scipy.special.**assoc_laguerre**(*x*, *n*, *k=0.0*)

Compute the generalized (associated) Laguerre polynomial of degree n and order k.

The polynomial $L_n^{(k)}(x)$ is orthogonal over `[0, inf)`, with weighting function `exp(-x) * x**k` with `k > -1`.

Notes

assoc_laguerre is a simple wrapper around *eval_genlaguerre*, with reversed argument order `(x, n, k=0.0) --> (n, k, x)`.

scipy.special.**eval_legendre**(*n*, *x*, *out=None*) **= <ufunc 'eval_legendre'>**

Evaluate Legendre polynomial at a point.

The Legendre polynomials can be defined via the Gauss hypergeometric function ${}_2F_1$ as

$$P_n(x) = {}_2F_1(-n, n+1; 1; (1-x)/2).$$

When n is an integer the result is a polynomial of degree n.

Parameters **n** : array_like

Degree of the polynomial. If not an integer, the result is determined via the relation to the Gauss hypergeometric function.

x : array_like

Points at which to evaluate the Legendre polynomial

Returns **P** : ndarray

Values of the Legendre polynomial

See also:

roots_legendre roots and quadrature weights of Legendre polynomials

legendre Legendre polynomial object

hyp2f1 Gauss hypergeometric function

numpy.polynomial.legendre.Legendre Legendre series

scipy.special.**eval_chebyt**(*n*, *x*, *out=None*) **= <ufunc 'eval_chebyt'>**

Evaluate Chebyshev polynomial of the first kind at a point.

The Chebyshev polynomials of the first kind can be defined via the Gauss hypergeometric function ${}_2F_1$ as

$$T_n(x) = {}_2F_1(n, -n; 1/2; (1-x)/2).$$

When n is an integer the result is a polynomial of degree n.

Parameters

n : array_like
: Degree of the polynomial. If not an integer, the result is determined via the relation to the Gauss hypergeometric function.

x : array_like
: Points at which to evaluate the Chebyshev polynomial

Returns

T : ndarray
: Values of the Chebyshev polynomial

See also:

roots_chebyt
roots and quadrature weights of Chebyshev polynomials of the first kind

chebyu Chebychev polynomial object

eval_chebyu
evaluate Chebyshev polynomials of the second kind

hyp2f1 Gauss hypergeometric function

numpy.polynomial.chebyshev.Chebyshev
Chebyshev series

Notes

This routine is numerically stable for *x* in `[-1, 1]` at least up to order `10000`.

`scipy.special.`**eval_chebyu**(*n*, *x*, *out=None*) = **<ufunc 'eval_chebyu'>**

Evaluate Chebyshev polynomial of the second kind at a point.

The Chebyshev polynomials of the second kind can be defined via the Gauss hypergeometric function ${}_2F_1$ as

$$U_n(x) = (n+1) {}_2F_1(-n, n+2; 3/2; (1-x)/2).$$

When n is an integer the result is a polynomial of degree n.

Parameters

n : array_like
: Degree of the polynomial. If not an integer, the result is determined via the relation to the Gauss hypergeometric function.

x : array_like
: Points at which to evaluate the Chebyshev polynomial

Returns

U : ndarray
: Values of the Chebyshev polynomial

See also:

roots_chebyu
roots and quadrature weights of Chebyshev polynomials of the second kind

chebyu Chebyshev polynomial object

eval_chebyt
evaluate Chebyshev polynomials of the first kind

hyp2f1 Gauss hypergeometric function

`scipy.special.`**eval_chebyc**(*n*, *x*, *out=None*) = **<ufunc 'eval_chebyc'>**

Evaluate Chebyshev polynomial of the first kind on [-2, 2] at a point.

These polynomials are defined as

$$S_n(x) = T_n(x/2)$$

where T_n is a Chebyshev polynomial of the first kind.

Parameters

n : array_like
: Degree of the polynomial. If not an integer, the result is determined via the relation to `eval_chebyt`.

Returns

x : array_like
Points at which to evaluate the Chebyshev polynomial

C : ndarray
Values of the Chebyshev polynomial

See also:

roots_chebyc
roots and quadrature weights of Chebyshev polynomials of the first kind on [-2, 2]

chebyc Chebyshev polynomial object

numpy.polynomial.chebyshev.Chebyshev
Chebyshev series

eval_chebyt
evaluate Chebycshev polynomials of the first kind

scipy.special.**eval_chebys**(*n*, *x*, *out=None*) = **<ufunc 'eval_chebys'>**

Evaluate Chebyshev polynomial of the second kind on [-2, 2] at a point.

These polynomials are defined as

$$S_n(x) = U_n(x/2)$$

where U_n is a Chebyshev polynomial of the second kind.

Parameters

n : array_like
Degree of the polynomial. If not an integer, the result is determined via the relation to *eval_chebyu*.

x : array_like
Points at which to evaluate the Chebyshev polynomial

Returns

S : ndarray
Values of the Chebyshev polynomial

See also:

roots_chebys
roots and quadrature weights of Chebyshev polynomials of the second kind on [-2, 2]

chebys Chebyshev polynomial object

eval_chebyu
evaluate Chebyshev polynomials of the second kind

scipy.special.**eval_jacobi**(*n*, *alpha*, *beta*, *x*, *out=None*) = **<ufunc 'eval_jacobi'>**

Evaluate Jacobi polynomial at a point.

The Jacobi polynomials can be defined via the Gauss hypergeometric function ${}_2F_1$ as

$$P_n^{(\alpha,\beta)}(x) = \frac{(\alpha+1)_n}{\Gamma(n+1)} {}_2F_1(-n, 1+\alpha+\beta+n; \alpha+1; (1-z)/2)$$

where $(\cdot)_n$ is the Pochhammer symbol; see *poch*. When n is an integer the result is a polynomial of degree n.

Parameters

n : array_like
Degree of the polynomial. If not an integer the result is determined via the relation to the Gauss hypergeometric function.

alpha : array_like
Parameter

beta : array_like
Parameter

x : array_like
Points at which to evaluate the polynomial

Returns

P : ndarray
Values of the Jacobi polynomial

See also:

roots_jacobi
roots and quadrature weights of Jacobi polynomials

jacobi Jacobi polynomial object

hyp2f1 Gauss hypergeometric function

scipy.special.**eval_laguerre**(*n*, *x*, *out=None*) = <ufunc 'eval_laguerre'>

Evaluate Laguerre polynomial at a point.

The Laguerre polynomials can be defined via the confluent hypergeometric function ${}_1F_1$ as

$$L_n(x) = {}_1F_1(-n, 1, x).$$

When n is an integer the result is a polynomial of degree n.

Parameters **n** : array_like

Degree of the polynomial. If not an integer the result is determined via the relation to the confluent hypergeometric function.

x : array_like

Points at which to evaluate the Laguerre polynomial

Returns **L** : ndarray

Values of the Laguerre polynomial

See also:

roots_laguerre
roots and quadrature weights of Laguerre polynomials

laguerre Laguerre polynomial object

numpy.polynomial.laguerre.Laguerre
Laguerre series

eval_genlaguerre
evaluate generalized Laguerre polynomials

scipy.special.**eval_genlaguerre**(*n*, *alpha*, *x*, *out=None*) = <ufunc 'eval_genlaguerre'>

Evaluate generalized Laguerre polynomial at a point.

The generalized Laguerre polynomials can be defined via the confluent hypergeometric function ${}_1F_1$ as

$$L_n^{(\alpha)}(x) = \binom{n+\alpha}{n} {}_1F_1(-n, \alpha+1, x).$$

When n is an integer the result is a polynomial of degree n. The Laguerre polynomials are the special case where $\alpha = 0$.

Parameters **n** : array_like

Degree of the polynomial. If not an integer the result is determined via the relation to the confluent hypergeometric function.

alpha : array_like

Parameter; must have `alpha > -1`

x : array_like

Points at which to evaluate the generalized Laguerre polynomial

Returns **L** : ndarray

Values of the generalized Laguerre polynomial

See also:

roots_genlaguerre
roots and quadrature weights of generalized Laguerre polynomials

genlaguerre
generalized Laguerre polynomial object

hyp1f1 confluent hypergeometric function

eval_laguerre evaluate Laguerre polynomials

scipy.special.**eval_hermite**(*n*, *x*, *out=None*) = <ufunc 'eval_hermite'>

Evaluate physicist's Hermite polynomial at a point.

Defined by

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2};$$

H_n is a polynomial of degree n.

Parameters

n : array_like
Degree of the polynomial

x : array_like
Points at which to evaluate the Hermite polynomial

Returns

H : ndarray
Values of the Hermite polynomial

See also:

roots_hermite roots and quadrature weights of physicist's Hermite polynomials

hermite physicist's Hermite polynomial object

numpy.polynomial.hermite.Hermite Physicist's Hermite series

eval_hermitenorm evaluate Probabilist's Hermite polynomials

scipy.special.**eval_hermitenorm**(*n*, *x*, *out=None*) = <ufunc 'eval_hermitenorm'>

Evaluate probabilist's (normalized) Hermite polynomial at a point.

Defined by

$$He_n(x) = (-1)^n e^{x^2/2} \frac{d^n}{dx^n} e^{-x^2/2};$$

He_n is a polynomial of degree n.

Parameters

n : array_like
Degree of the polynomial

x : array_like
Points at which to evaluate the Hermite polynomial

Returns

He : ndarray
Values of the Hermite polynomial

See also:

roots_hermitenorm roots and quadrature weights of probabilist's Hermite polynomials

hermitenorm probabilist's Hermite polynomial object

numpy.polynomial.hermite_e.HermiteE Probabilist's Hermite series

eval_hermite evaluate physicist's Hermite polynomials

scipy.special.**eval_gegenbauer**(*n*, *alpha*, *x*, *out=None*) = <ufunc 'eval_gegenbauer'>

Evaluate Gegenbauer polynomial at a point.

The Gegenbauer polynomials can be defined via the Gauss hypergeometric function ${}_2F_1$ as

$$C_n^{(\alpha)} = \frac{(2\alpha)_n}{\Gamma(n+1)} {}_2F_1(-n, 2\alpha+n; \alpha+1/2; (1-z)/2).$$

When n is an integer the result is a polynomial of degree n.

Parameters **n** : array_like
Degree of the polynomial. If not an integer, the result is determined via the relation to the Gauss hypergeometric function.

alpha : array_like
Parameter

x : array_like
Points at which to evaluate the Gegenbauer polynomial

Returns **C** : ndarray
Values of the Gegenbauer polynomial

See also:

roots_gegenbauer
roots and quadrature weights of Gegenbauer polynomials

gegenbauer Gegenbauer polynomial object

hyp2f1 Gauss hypergeometric function

scipy.special.**eval_sh_legendre**(*n*, *x*, *out=None*) **= <ufunc 'eval_sh_legendre'>**

Evaluate shifted Legendre polynomial at a point.

These polynomials are defined as

$$P_n^*(x) = P_n(2x-1)$$

where P_n is a Legendre polynomial.

Parameters **n** : array_like
Degree of the polynomial. If not an integer, the value is determined via the relation to *eval_legendre*.

x : array_like
Points at which to evaluate the shifted Legendre polynomial

Returns **P** : ndarray
Values of the shifted Legendre polynomial

See also:

roots_sh_legendre
roots and quadrature weights of shifted Legendre polynomials

sh_legendre
shifted Legendre polynomial object

eval_legendre
evaluate Legendre polynomials

numpy.polynomial.legendre.Legendre
Legendre series

scipy.special.**eval_sh_chebyt**(*n*, *x*, *out=None*) **= <ufunc 'eval_sh_chebyt'>**

Evaluate shifted Chebyshev polynomial of the first kind at a point.

These polynomials are defined as

$$T_n^*(x) = T_n(2x-1)$$

where T_n is a Chebyshev polynomial of the first kind.

Parameters **n** : array_like

Degree of the polynomial. If not an integer, the result is determined via the relation to *eval_chebyt*.

x : array_like

Points at which to evaluate the shifted Chebyshev polynomial

Returns **T** : ndarray

Values of the shifted Chebyshev polynomial

See also:

roots_sh_chebyt roots and quadrature weights of shifted Chebyshev polynomials of the first kind

sh_chebyt shifted Chebyshev polynomial object

eval_chebyt evalaute Chebyshev polynomials of the first kind

numpy.polynomial.chebyshev.Chebyshev Chebyshev series

scipy.special.**eval_sh_chebyu**(*n*, *x*, *out=None*) **= <ufunc 'eval_sh_chebyu'>**

Evaluate shifted Chebyshev polynomial of the second kind at a point.

These polynomials are defined as

$$U_n^*(x) = U_n(2x - 1)$$

where U_n is a Chebyshev polynomial of the first kind.

Parameters **n** : array_like

Degree of the polynomial. If not an integer, the result is determined via the relation to *eval_chebyu*.

x : array_like

Points at which to evaluate the shifted Chebyshev polynomial

Returns **U** : ndarray

Values of the shifted Chebyshev polynomial

See also:

roots_sh_chebyu roots and quadrature weights of shifted Chebychev polynomials of the second kind

sh_chebyu shifted Chebyshev polynomial object

eval_chebyu evaluate Chebyshev polynomials of the second kind

scipy.special.**eval_sh_jacobi**(*n*, *p*, *q*, *x*, *out=None*) **= <ufunc 'eval_sh_jacobi'>**

Evaluate shifted Jacobi polynomial at a point.

Defined by

$$G_n^{(p,q)}(x) = \binom{2n + p - 1}{n}^{-1} P_n^{(p-q,q-1)}(2x - 1),$$

where $P_n^{(\cdot,\cdot)}$ is the n-th Jacobi polynomial.

Parameters **n** : int

Degree of the polynomial. If not an integer, the result is determined via the relation to *binom* and *eval_jacobi*.

p : float

Parameter

q : float

Parameter

Returns **G** : ndarray

Values of the shifted Jacobi polynomial.

See also:

roots_sh_jacobi
roots and quadrature weights of shifted Jacobi polynomials

sh_jacobi shifted Jacobi polynomial object

eval_jacobi
evaluate Jacobi polynomials

The following functions compute roots and quadrature weights for orthogonal polynomials:

roots_legendre(n[, mu])	Gauss-Legendre quadrature.
roots_chebyt(n[, mu])	Gauss-Chebyshev (first kind) quadrature.
roots_chebyu(n[, mu])	Gauss-Chebyshev (second kind) quadrature.
roots_chebyc(n[, mu])	Gauss-Chebyshev (first kind) quadrature.
roots_chebys(n[, mu])	Gauss-Chebyshev (second kind) quadrature.
roots_jacobi(n, alpha, beta[, mu])	Gauss-Jacobi quadrature.
roots_laguerre(n[, mu])	Gauss-Laguerre quadrature.
roots_genlaguerre(n, alpha[, mu])	Gauss-generalized Laguerre quadrature.
roots_hermite(n[, mu])	Gauss-Hermite (physicst's) quadrature.
roots_hermitenorm(n[, mu])	Gauss-Hermite (statistician's) quadrature.
roots_gegenbauer(n, alpha[, mu])	Gauss-Gegenbauer quadrature.
roots_sh_legendre(n[, mu])	Gauss-Legendre (shifted) quadrature.
roots_sh_chebyt(n[, mu])	Gauss-Chebyshev (first kind, shifted) quadrature.
roots_sh_chebyu(n[, mu])	Gauss-Chebyshev (second kind, shifted) quadrature.
roots_sh_jacobi(n, p1, q1[, mu])	Gauss-Jacobi (shifted) quadrature.

scipy.special.**roots_legendre**(*n*, *mu=False*)

Gauss-Legendre quadrature.

Computes the sample points and weights for Gauss-Legendre quadrature. The sample points are the roots of the n-th degree Legendre polynomial $P_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[-1, 1]$ with weight function $f(x) = 1.0$.

Parameters **n** : int
quadrature order

mu : bool, optional
If True, return the sum of the weights, optional.

Returns **x** : ndarray
Sample points

w : ndarray
Weights

mu : float
Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*, numpy.polynomial.legendre.leggauss

scipy.special.**roots_chebyt**(*n*, *mu=False*)

Gauss-Chebyshev (first kind) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree Chebyshev polynomial of the first kind, $T_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[-1, 1]$ with weight function $f(x) = 1/\sqrt{1 - x^2}$.

Parameters **n** : int
quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*, numpy.polynomial.chebyshev.chebgauss

`scipy.special.`**`roots_chebyu`**(*n*, *mu=False*)

Gauss-Chebyshev (second kind) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree Chebyshev polynomial of the second kind, $U_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[-1,1]$ with weight function $f(x)=\sqrt{1-x^2}$.

Parameters **n** : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

`scipy.special.`**`roots_chebyc`**(*n*, *mu=False*)

Gauss-Chebyshev (first kind) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree Chebyshev polynomial of the first kind, $C_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[-2,2]$ with weight function $f(x)=1/\sqrt{1-(x/2)^2}$.

Parameters **n** : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

`scipy.special.`**`roots_chebys`**(*n*, *mu=False*)

Gauss-Chebyshev (second kind) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree Chebyshev polynomial of the second kind, $S_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[-2, 2]$ with weight function $f(x) = \sqrt{1-(x/2)^2}$.

Parameters **n** : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_jacobi**(*n*, *alpha*, *beta*, *mu=False*)

Gauss-Jacobi quadrature.

Computes the sample points and weights for Gauss-Jacobi quadrature. The sample points are the roots of the n-th degree Jacobi polynomial, $P_n^{\alpha,\beta}(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[-1, 1]$ with weight function $f(x) = (1-x)^{\alpha}(1+x)^{\beta}$.

Parameters **n** : int
: quadrature order

alpha : float
: alpha must be > -1

beta : float
: beta must be > 0

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_laguerre**(*n*, *mu=False*)

Gauss-Laguerre quadrature.

Computes the sample points and weights for Gauss-Laguerre quadrature. The sample points are the roots of the n-th degree Laguerre polynomial, $L_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n-1$ or less over the interval $[0, \infty]$ with weight function $f(x) = e^{-x}$.

Parameters **n** : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns **x** : ndarray
: Sample points

w : ndarray
: Weights

mu : float

Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*, numpy.polynomial.laguerre.laggauss

scipy.special.**roots_genlaguerre**(*n*, *alpha*, *mu=False*)

Gauss-generalized Laguerre quadrature.

Computes the sample points and weights for Gauss-generalized Laguerre quadrature. The sample points are the roots of the n-th degree generalized Laguerre polynomial, $L_n^\alpha(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[0, \infty]$ with weight function $f(x) = x^\alpha e^{-x}$.

Parameters

n : int

quadrature order

alpha : float

alpha must be > -1

mu : bool, optional

If True, return the sum of the weights, optional.

Returns

x : ndarray

Sample points

w : ndarray

Weights

mu : float

Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_hermite**(*n*, *mu=False*)

Gauss-Hermite (physicst's) quadrature.

Computes the sample points and weights for Gauss-Hermite quadrature. The sample points are the roots of the n-th degree Hermite polynomial, $H_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[-\infty, \infty]$ with weight function $f(x) = e^{-x^2}$.

Parameters

n : int

quadrature order

mu : bool, optional

If True, return the sum of the weights, optional.

Returns

x : ndarray

Sample points

w : ndarray

Weights

mu : float

Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*, numpy.polynomial.hermite.hermgauss, *roots_hermitenorm*

Notes

For small n up to 150 a modified version of the Golub-Welsch algorithm is used. Nodes are computed from the eigenvalue problem and improved by one step of a Newton iteration. The weights are computed from the well-known analytical formula.

For n larger than 150 an optimal asymptotic algorithm is applied which computes nodes and weights in a numerically stable manner. The algorithm has linear runtime making computation for very large n (several thousand or more) feasible.

References

[townsend.trogdon.olver-2014], [townsend.trogdon.olver-2015]

`scipy.special.`**`roots_hermitenorm`**(*n*, *mu=False*)

Gauss-Hermite (statistician's) quadrature.

Computes the sample points and weights for Gauss-Hermite quadrature. The sample points are the roots of the n-th degree Hermite polynomial, $He_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[-\infty, \infty]$ with weight function $f(x) = e^{-x^2/2}$.

Parameters

n : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns

x : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

`scipy.integrate.quadrature`, *`scipy.integrate.fixed_quad`*, `numpy.polynomial.hermite_e.hermegauss`

Notes

For small n up to 150 a modified version of the Golub-Welsch algorithm is used. Nodes are computed from the eigenvalue problem and improved by one step of a Newton iteration. The weights are computed from the well-known analytical formula.

For n larger than 150 an optimal asymptotic algorithm is used which computes nodes and weights in a numerical stable manner. The algorithm has linear runtime making computation for very large n (several thousand or more) feasible.

`scipy.special.`**`roots_gegenbauer`**(*n*, *alpha*, *mu=False*)

Gauss-Gegenbauer quadrature.

Computes the sample points and weights for Gauss-Gegenbauer quadrature. The sample points are the roots of the n-th degree Gegenbauer polynomial, $C_n^\alpha(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[-1, 1]$ with weight function $f(x) = (1 - x^2)^{\alpha - 1/2}$.

Parameters

n : int
: quadrature order

alpha : float
: alpha must be > -0.5

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns

x : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_sh_legendre**(*n*, *mu=False*)

Gauss-Legendre (shifted) quadrature.

Computes the sample points and weights for Gauss-Legendre quadrature. The sample points are the roots of the n-th degree shifted Legendre polynomial $P_n^*(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[0, 1]$ with weight function $f(x) = 1.0$.

Parameters

n : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns

x : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_sh_chebyt**(*n*, *mu=False*)

Gauss-Chebyshev (first kind, shifted) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree shifted Chebyshev polynomial of the first kind, $T_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[0, 1]$ with weight function $f(x) = 1/\sqrt{x - x^2}$.

Parameters

n : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns

x : ndarray
: Sample points

w : ndarray
: Weights

mu : float
: Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_sh_chebyu**(*n*, *mu=False*)

Gauss-Chebyshev (second kind, shifted) quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. The sample points are the roots of the n-th degree shifted Chebyshev polynomial of the second kind, $U_n(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[0, 1]$ with weight function $f(x) = \sqrt{x - x^2}$.

Parameters

n : int
: quadrature order

mu : bool, optional
: If True, return the sum of the weights, optional.

Returns

x : ndarray
: Sample points

w : ndarray

Weights

mu : float

Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

scipy.special.**roots_sh_jacobi**(*n*, *p1*, *q1*, *mu=False*)

Gauss-Jacobi (shifted) quadrature.

Computes the sample points and weights for Gauss-Jacobi (shifted) quadrature. The sample points are the roots of the n-th degree shifted Jacobi polynomial, $G_n^{p,q}(x)$. These sample points and weights correctly integrate polynomials of degree $2n - 1$ or less over the interval $[0, 1]$ with weight function $f(x) = (1 - x)^{p-q} x^{q-1}$

Parameters

n : int

quadrature order

p1 : float

(p1 - q1) must be > -1

q1 : float

q1 must be > 0

mu : bool, optional

If True, return the sum of the weights, optional.

Returns

x : ndarray

Sample points

w : ndarray

Weights

mu : float

Sum of the weights

See also:

scipy.integrate.quadrature, *scipy.integrate.fixed_quad*

The functions below, in turn, return the polynomial coefficients in `orthopoly1d` objects, which function similarly as numpy.poly1d. The `orthopoly1d` class also has an attribute `weights` which returns the roots, weights, and total weights for the appropriate form of Gaussian quadrature. These are returned in an `n x 3` array with roots in the first column, weights in the second column, and total weights in the final column. Note that `orthopoly1d` objects are converted to `poly1d` when doing arithmetic, and lose information of the original orthogonal polynomial.

legendre(n[, monic])	Legendre polynomial.
chebyt(n[, monic])	Chebyshev polynomial of the first kind.
chebyu(n[, monic])	Chebyshev polynomial of the second kind.
chebyc(n[, monic])	Chebyshev polynomial of the first kind on $[-2, 2]$.
chebys(n[, monic])	Chebyshev polynomial of the second kind on $[-2, 2]$.
jacobi(n, alpha, beta[, monic])	Jacobi polynomial.
laguerre(n[, monic])	Laguerre polynomial.
genlaguerre(n, alpha[, monic])	Generalized (associated) Laguerre polynomial.
hermite(n[, monic])	Physicist's Hermite polynomial.
hermitenorm(n[, monic])	Normalized (probabilist's) Hermite polynomial.
gegenbauer(n, alpha[, monic])	Gegenbauer (ultraspherical) polynomial.
sh_legendre(n[, monic])	Shifted Legendre polynomial.
sh_chebyt(n[, monic])	Shifted Chebyshev polynomial of the first kind.
sh_chebyu(n[, monic])	Shifted Chebyshev polynomial of the second kind.
sh_jacobi(n, p, q[, monic])	Shifted Jacobi polynomial.

scipy.special.**legendre**(*n*, *monic=False*)

Legendre polynomial.

Defined to be the solution of

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right]+n(n+1)P_n(x)=0;$$

$P_n(x)$ is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **P** : orthopoly1d

Legendre polynomial.

Notes

The polynomials P_n are orthogonal over $[-1, 1]$ with weight function 1.

Examples

Generate the 3rd-order Legendre polynomial 1/2*(5x^3 + 0x^2 - 3x + 0):

```
>>> from scipy.special import legendre
>>> legendre(3)
poly1d([ 2.5,  0. , -1.5,  0. ])
```

`scipy.special.`**`chebyt`**(*n*, *monic=False*)

Chebyshev polynomial of the first kind.

Defined to be the solution of

$$(1-x^2)\frac{d^2}{dx^2}T_n-x\frac{d}{dx}T_n+n^2T_n=0;$$

T_n is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **T** : orthopoly1d

Chebyshev polynomial of the first kind.

See also:

`chebyu` Chebyshev polynomial of the second kind.

Notes

The polynomials T_n are orthogonal over $[-1, 1]$ with weight function $(1-x^2)^{-1/2}$.

`scipy.special.`**`chebyu`**(*n*, *monic=False*)

Chebyshev polynomial of the second kind.

Defined to be the solution of

$$(1-x^2)\frac{d^2}{dx^2}U_n-3x\frac{d}{dx}U_n+n(n+2)U_n=0;$$

U_n is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **U** : orthopoly1d

Chebyshev polynomial of the second kind.

See also:

chebyt Chebyshev polynomial of the first kind.

Notes

The polynomials U_n are orthogonal over $[-1, 1]$ with weight function $(1 - x^2)^{1/2}$.

`scipy.special.chebyc(n, monic=False)`

Chebyshev polynomial of the first kind on $[-2, 2]$.

Defined as $C_n(x) = 2T_n(x/2)$, where T_n is the nth Chebychev polynomial of the first kind.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **C** : orthopoly1d

Chebyshev polynomial of the first kind on $[-2, 2]$.

See also:

chebyt Chebyshev polynomial of the first kind.

Notes

The polynomials $C_n(x)$ are orthogonal over $[-2, 2]$ with weight function $1/\sqrt{1 - (x/2)^2}$.

References

[R390]

`scipy.special.chebys(n, monic=False)`

Chebyshev polynomial of the second kind on $[-2, 2]$.

Defined as $S_n(x) = U_n(x/2)$ where U_n is the nth Chebychev polynomial of the second kind.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **S** : orthopoly1d

Chebyshev polynomial of the second kind on $[-2, 2]$.

See also:

chebyu Chebyshev polynomial of the second kind

Notes

The polynomials $S_n(x)$ are orthogonal over $[-2, 2]$ with weight function $\sqrt{1 - (x/2)}^2$.

References

[R391]

`scipy.special.jacobi(n, alpha, beta, monic=False)`
Jacobi polynomial.

Defined to be the solution of

$$(1-x^2)\frac{d^2}{dx^2}P_n^{(\alpha,\beta)} + (\beta-\alpha-(\alpha+\beta+2)x)\frac{d}{dx}P_n^{(\alpha,\beta)} + n(n+\alpha+\beta+1)P_n^{(\alpha,\beta)} = 0$$

for $\alpha, \beta > -1$; $P_n^{(\alpha,\beta)}$ is a polynomial of degree n.

Parameters **n** : int
Degree of the polynomial.
alpha : float
Parameter, must be greater than -1.
beta : float
Parameter, must be greater than -1.
monic : bool, optional
If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **P** : orthopoly1d
Jacobi polynomial.

Notes

For fixed α, β, the polynomials $P_n^{(\alpha,\beta)}$ are orthogonal over $[-1, 1]$ with weight function $(1-x)^\alpha(1+x)^\beta$.

`scipy.special.laguerre(n, monic=False)`
Laguerre polynomial.

Defined to be the solution of

$$x\frac{d^2}{dx^2}L_n + (1-x)\frac{d}{dx}L_n + nL_n = 0;$$

L_n is a polynomial of degree n.

Parameters **n** : int
Degree of the polynomial.
monic : bool, optional
If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **L** : orthopoly1d
Laguerre Polynomial.

Notes

The polynomials L_n are orthogonal over $[0, \infty)$ with weight function e^{-x}.

`scipy.special.genlaguerre(n, alpha, monic=False)`
Generalized (associated) Laguerre polynomial.

Defined to be the solution of

$$x\frac{d^2}{dx^2}L_n^{(\alpha)} + (\alpha+1-x)\frac{d}{dx}L_n^{(\alpha)} + nL_n^{(\alpha)} = 0,$$

where $\alpha > -1$; $L_n^{(\alpha)}$ is a polynomial of degree n.

Parameters **n** : int
Degree of the polynomial.
alpha : float
Parameter, must be greater than -1.
monic : bool, optional
If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **L** : orthopoly1d

Generalized Laguerre polynomial.

See also:

laguerre Laguerre polynomial.

Notes

For fixed α, the polynomials $L_n^{(\alpha)}$ are orthogonal over $[0, \infty)$ with weight function $e^{-x}x^\alpha$.

The Laguerre polynomials are the special case where $\alpha = 0$.

`scipy.special.hermite(n, monic=False)`

Physicist's Hermite polynomial.

Defined by

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2};$$

H_n is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **H** : orthopoly1d

Hermite polynomial.

Notes

The polynomials H_n are orthogonal over $(-\infty, \infty)$ with weight function e^{-x^2}.

`scipy.special.hermitenorm(n, monic=False)`

Normalized (probabilist's) Hermite polynomial.

Defined by

$$He_n(x) = (-1)^n e^{x^2/2} \frac{d^n}{dx^n} e^{-x^2/2};$$

He_n is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **He** : orthopoly1d

Hermite polynomial.

Notes

The polynomials He_n are orthogonal over $(-\infty, \infty)$ with weight function $e^{-x^2/2}$.

`scipy.special.gegenbauer(n, alpha, monic=False)`

Gegenbauer (ultraspherical) polynomial.

Defined to be the solution of

$$(1 - x^2)\frac{d^2}{dx^2}C_n^{(\alpha)} - (2\alpha + 1)x\frac{d}{dx}C_n^{(\alpha)} + n(n + 2\alpha)C_n^{(\alpha)} = 0$$

for $\alpha > -1/2$; $C_n^{(\alpha)}$ is a polynomial of degree n.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **C** : orthopoly1d

Gegenbauer polynomial.

Notes

The polynomials $C_n^{(\alpha)}$ are orthogonal over $[-1, 1]$ with weight function $(1 - x^2)^{(\alpha - 1/2)}$.

`scipy.special.`**`sh_legendre`**(*n*, *monic=False*)

Shifted Legendre polynomial.

Defined as $P_n^*(x) = P_n(2x - 1)$ for P_n the nth Legendre polynomial.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **P** : orthopoly1d

Shifted Legendre polynomial.

Notes

The polynomials P_n^* are orthogonal over $[0, 1]$ with weight function 1.

`scipy.special.`**`sh_chebyt`**(*n*, *monic=False*)

Shifted Chebyshev polynomial of the first kind.

Defined as $T_n^*(x) = T_n(2x - 1)$ for T_n the nth Chebyshev polynomial of the first kind.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **T** : orthopoly1d

Shifted Chebyshev polynomial of the first kind.

Notes

The polynomials T_n^* are orthogonal over $[0, 1]$ with weight function $(x - x^2)^{-1/2}$.

`scipy.special.`**`sh_chebyu`**(*n*, *monic=False*)

Shifted Chebyshev polynomial of the second kind.

Defined as $U_n^*(x) = U_n(2x - 1)$ for U_n the nth Chebyshev polynomial of the second kind.

Parameters **n** : int

Degree of the polynomial.

monic : bool, optional

If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns **U** : orthopoly1d

Shifted Chebyshev polynomial of the second kind.

Notes

The polynomials U_n^* are orthogonal over $[0, 1]$ with weight function $(x - x^2)^{1/2}$.

`scipy.special.`**`sh_jacobi`**(*n*, *p*, *q*, *monic=False*)

Shifted Jacobi polynomial.

Defined by

$$G_n^{(p,q)}(x) = \binom{2n+p-1}{n}^{-1} P_n^{(p-q,q-1)}(2x-1),$$

where $P_n^{(\cdot,\cdot)}$ is the nth Jacobi polynomial.

Parameters
- **n** : int
  Degree of the polynomial.
- **p** : float
  Parameter, must have $p > q - 1$.
- **q** : float
  Parameter, must be greater than 0.
- **monic** : bool, optional
  If *True*, scale the leading coefficient to be 1. Default is *False*.

Returns
- **G** : orthopoly1d
  Shifted Jacobi polynomial.

Notes

For fixed p, q, the polynomials $G_n^{(p,q)}$ are orthogonal over $[0, 1]$ with weight function $(1 - x)^{p-q} x^{q-1}$.

> **Warning:** Computing values of high-order polynomials (around `order > 20`) using polynomial coefficients is numerically unstable. To evaluate polynomial values, the `eval_*` functions should be used instead.

Hypergeometric Functions

hyp2f1(a, b, c, z)	Gauss hypergeometric function 2F1(a, b; c; z).
hyp1f1(a, b, x)	Confluent hypergeometric function 1F1(a, b; x)
hyperu(a, b, x)	Confluent hypergeometric function U(a, b, x) of the second kind
hyp0f1(v, x)	Confluent hypergeometric limit function 0F1.
hyp2f0(a, b, x, type)	Hypergeometric function 2F0 in y and an error estimate
hyp1f2(a, b, c, x)	Hypergeometric function 1F2 and error estimate
hyp3f0(a, b, c, x)	Hypergeometric function 3F0 in y and an error estimate

`scipy.special.`**`hyp2f1`**(*a, b, c, z*) **= <ufunc 'hyp2f1'>**
Gauss hypergeometric function 2F1(a, b; c; z).

`scipy.special.`**`hyp1f1`**(*a, b, x*) **= <ufunc 'hyp1f1'>**
Confluent hypergeometric function 1F1(a, b; x)

`scipy.special.`**`hyperu`**(*a, b, x*) **= <ufunc 'hyperu'>**
Confluent hypergeometric function U(a, b, x) of the second kind

`scipy.special.`**`hyp0f1`**(*v, x*) **= <ufunc 'hyp0f1'>**
Confluent hypergeometric limit function 0F1.

Parameters
- **v, z** : array_like
  Input values.

Returns
- **hyp0f1** : ndarray
  The confluent hypergeometric limit function.

Notes

This function is defined as:

$$_0F_1(v,z) = \sum_{k=0}^{\infty} \frac{z^k}{(v)_k k!}.$$

It's also the limit as $q \to \infty$ of $_1F_1(q;v;z/q)$, and satisfies the differential equation $f''(z) + vf'(z) = f(z)$.

`scipy.special.hyp2f0(a, b, x, type)` = **<ufunc 'hyp2f0'>**

Hypergeometric function 2F0 in y and an error estimate

The parameter *type* determines a convergence factor and can be either 1 or 2.

Returns y

Value of the function

err

Error estimate

`scipy.special.hyp1f2(a, b, c, x)` = **<ufunc 'hyp1f2'>**

Hypergeometric function 1F2 and error estimate

Returns y

Value of the function

err

Error estimate

`scipy.special.hyp3f0(a, b, c, x)` = **<ufunc 'hyp3f0'>**

Hypergeometric function 3F0 in y and an error estimate

Returns y

Value of the function

err

Error estimate

Parabolic Cylinder Functions

pbdv(v, x)	Parabolic cylinder function D
pbvv(v, x)	Parabolic cylinder function V
pbwa(a, x)	Parabolic cylinder function W

`scipy.special.pbdv(v, x)` = **<ufunc 'pbdv'>**

Parabolic cylinder function D

Returns (d, dp) the parabolic cylinder function Dv(x) in d and the derivative, Dv'(x) in dp.

Returns d

Value of the function

dp

Value of the derivative vs x

`scipy.special.pbvv(v, x)` = **<ufunc 'pbvv'>**

Parabolic cylinder function V

Returns the parabolic cylinder function Vv(x) in v and the derivative, Vv'(x) in vp.

Returns v

Value of the function

vp
Value of the derivative vs x

scipy.special.**pbwa**(*a*, *x*) = <ufunc 'pbwa'>
Parabolic cylinder function W

Returns the parabolic cylinder function W(a, x) in w and the derivative, W'(a, x) in wp.

Warning: May not be accurate for large (>5) arguments in a and/or x.

Returns w
Value of the function
wp
Value of the derivative vs x

These are not universal functions:

pbdv_seq(v, x)	Parabolic cylinder functions Dv(x) and derivatives.
pbvv_seq(v, x)	Parabolic cylinder functions Vv(x) and derivatives.
pbdn_seq(n, z)	Parabolic cylinder functions Dn(z) and derivatives.

scipy.special.**pbdv_seq**(*v*, *x*)
Parabolic cylinder functions Dv(x) and derivatives.

Parameters **v** : float
Order of the parabolic cylinder function
x : float
Value at which to evaluate the function and derivatives

Returns **dv** : ndarray
Values of D_vi(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.
dp : ndarray
Derivatives D_vi'(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.

References

[R505]

scipy.special.**pbvv_seq**(*v*, *x*)
Parabolic cylinder functions Vv(x) and derivatives.

Parameters **v** : float
Order of the parabolic cylinder function
x : float
Value at which to evaluate the function and derivatives

Returns **dv** : ndarray
Values of V_vi(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.
dp : ndarray
Derivatives V_vi'(x), for vi=v-int(v), vi=1+v-int(v), ..., vi=v.

References

[R506]

scipy.special.**pbdn_seq**(*n*, *z*)
Parabolic cylinder functions Dn(z) and derivatives.

Parameters **n** : int

		Order of the parabolic cylinder function
	z : complex	Value at which to evaluate the function and derivatives
Returns	**dv** : ndarray	Values of D_i(z), for i=0, ..., i=n.
	dp : ndarray	Derivatives D_i'(z), for i=0, ..., i=n.

References

[R504]

Mathieu and Related Functions

mathieu_a(m, q)	Characteristic value of even Mathieu functions
mathieu_b(m, q)	Characteristic value of odd Mathieu functions

`scipy.special.`**`mathieu_a`**(*m*, *q*) **= <ufunc 'mathieu_a'>**

Characteristic value of even Mathieu functions

Returns the characteristic value for the even solution, `ce_m(z, q)`, of Mathieu's equation.

`scipy.special.`**`mathieu_b`**(*m*, *q*) **= <ufunc 'mathieu_b'>**

Characteristic value of odd Mathieu functions

Returns the characteristic value for the odd solution, `se_m(z, q)`, of Mathieu's equation.

These are not universal functions:

mathieu_even_coef(m, q)	Fourier coefficients for even Mathieu and modified Mathieu functions.
mathieu_odd_coef(m, q)	Fourier coefficients for even Mathieu and modified Mathieu functions.

`scipy.special.`**`mathieu_even_coef`**(*m*, *q*)

Fourier coefficients for even Mathieu and modified Mathieu functions.

The Fourier series of the even solutions of the Mathieu differential equation are of the form

$$\mathrm{ce}_{2n}(z,q) = \sum_{k=0}^{\infty} A_{(2n)}^{(2k)} \cos 2kz$$

$$\mathrm{ce}_{2n+1}(z,q) = \sum_{k=0}^{\infty} A_{(2n+1)}^{(2k+1)} \cos(2k+1)z$$

This function returns the coefficients $A_{(2n)}^{(2k)}$ for even input m=2n, and the coefficients $A_{(2n+1)}^{(2k+1)}$ for odd input m=2n+1.

Parameters	**m** : int	Order of Mathieu functions. Must be non-negative.
	q : float (>=0)	Parameter of Mathieu functions. Must be non-negative.
Returns	**Ak** : ndarray	Even or odd Fourier coefficients, corresponding to even or odd m.

References

[R490], *[R491]*

scipy.special.**mathieu_odd_coef**(*m*, *q*)

Fourier coefficients for even Mathieu and modified Mathieu functions.

The Fourier series of the odd solutions of the Mathieu differential equation are of the form

$$\mathrm{se}_{2n+1}(z,q) = \sum_{k=0}^{\infty} B_{(2n+1)}^{(2k+1)} \sin(2k+1)z$$

$$\mathrm{se}_{2n+2}(z,q) = \sum_{k=0}^{\infty} B_{(2n+2)}^{(2k+2)} \sin(2k+2)z$$

This function returns the coefficients $B_{(2n+2)}^{(2k+2)}$ for even input m=2n+2, and the coefficients $B_{(2n+1)}^{(2k+1)}$ for odd input m=2n+1.

Parameters **m** : int

Order of Mathieu functions. Must be non-negative.

q : float (>=0)

Parameter of Mathieu functions. Must be non-negative.

Returns **Bk** : ndarray

Even or odd Fourier coefficients, corresponding to even or odd m.

References

[R492]

The following return both function and first derivative:

mathieu_cem(m, q, x)	Even Mathieu function and its derivative
mathieu_sem(m, q, x)	Odd Mathieu function and its derivative
mathieu_modcem1(m, q, x)	Even modified Mathieu function of the first kind and its derivative
mathieu_modcem2(m, q, x)	Even modified Mathieu function of the second kind and its derivative
mathieu_modsem1(m, q, x)	Odd modified Mathieu function of the first kind and its derivative
mathieu_modsem2(m, q, x)	Odd modified Mathieu function of the second kind and its derivative

scipy.special.**mathieu_cem**(*m*, *q*, *x*) **= <ufunc 'mathieu_cem'>**

Even Mathieu function and its derivative

Returns the even Mathieu function, `ce_m(x, q)`, of order *m* and parameter *q* evaluated at *x* (given in degrees). Also returns the derivative with respect to *x* of ce_m(x, q)

Parameters **m**

Order of the function

q

Parameter of the function

x

Argument of the function, *given in degrees, not radians*

Returns y

Value of the function

yp

Value of the derivative vs x

`scipy.special.mathieu_sem(m, q, x) = <ufunc 'mathieu_sem'>`

Odd Mathieu function and its derivative

Returns the odd Mathieu function, se_m(x, q), of order *m* and parameter *q* evaluated at *x* (given in degrees). Also returns the derivative with respect to *x* of se_m(x, q).

Parameters

m
: Order of the function

q
: Parameter of the function

x
: Argument of the function, *given in degrees, not radians*.

Returns

y
: Value of the function

yp
: Value of the derivative vs x

`scipy.special.mathieu_modcem1(m, q, x) = <ufunc 'mathieu_modcem1'>`

Even modified Mathieu function of the first kind and its derivative

Evaluates the even modified Mathieu function of the first kind, `Mc1m(x, q)`, and its derivative at *x* for order *m* and parameter *q*.

Returns

y
: Value of the function

yp
: Value of the derivative vs x

`scipy.special.mathieu_modcem2(m, q, x) = <ufunc 'mathieu_modcem2'>`

Even modified Mathieu function of the second kind and its derivative

Evaluates the even modified Mathieu function of the second kind, Mc2m(x, q), and its derivative at *x* (given in degrees) for order *m* and parameter *q*.

Returns

y
: Value of the function

yp
: Value of the derivative vs x

`scipy.special.mathieu_modsem1(m, q, x) = <ufunc 'mathieu_modsem1'>`

Odd modified Mathieu function of the first kind and its derivative

Evaluates the odd modified Mathieu function of the first kind, Ms1m(x, q), and its derivative at *x* (given in degrees) for order *m* and parameter *q*.

Returns

y
: Value of the function

yp
: Value of the derivative vs x

`scipy.special.mathieu_modsem2(m, q, x) = <ufunc 'mathieu_modsem2'>`

Odd modified Mathieu function of the second kind and its derivative

Evaluates the odd modified Mathieu function of the second kind, Ms2m(x, q), and its derivative at *x* (given in degrees) for order *m* and parameter q.

Returns

y
: Value of the function

yp
: Value of the derivative vs x

Spheroidal Wave Functions

pro_ang1(m, n, c, x)	Prolate spheroidal angular function of the first kind and its derivative
pro_rad1(m, n, c, x)	Prolate spheroidal radial function of the first kind and its derivative
pro_rad2(m, n, c, x)	Prolate spheroidal radial function of the secon kind and its derivative
obl_ang1(m, n, c, x)	Oblate spheroidal angular function of the first kind and its derivative
obl_rad1(m, n, c, x)	Oblate spheroidal radial function of the first kind and its derivative
obl_rad2(m, n, c, x)	Oblate spheroidal radial function of the second kind and its derivative.
pro_cv(m, n, c)	Characteristic value of prolate spheroidal function
obl_cv(m, n, c)	Characteristic value of oblate spheroidal function
pro_cv_seq(m, n, c)	Characteristic values for prolate spheroidal wave functions.
obl_cv_seq(m, n, c)	Characteristic values for oblate spheroidal wave functions.

scipy.special.**pro_ang1**(*m*, *n*, *c*, *x*) = <ufunc 'pro_ang1'>

Prolate spheroidal angular function of the first kind and its derivative

Computes the prolate spheroidal angular function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns **s**

Value of the function

sp

Value of the derivative vs x

scipy.special.**pro_rad1**(*m*, *n*, *c*, *x*) = <ufunc 'pro_rad1'>

Prolate spheroidal radial function of the first kind and its derivative

Computes the prolate spheroidal radial function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns **s**

Value of the function

sp

Value of the derivative vs x

scipy.special.**pro_rad2**(*m*, *n*, *c*, *x*) = <ufunc 'pro_rad2'>

Prolate spheroidal radial function of the secon kind and its derivative

Computes the prolate spheroidal radial function of the second kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns **s**

Value of the function

sp

Value of the derivative vs x

scipy.special.**obl_ang1**(*m*, *n*, *c*, *x*) = <ufunc 'obl_ang1'>

Oblate spheroidal angular function of the first kind and its derivative

Computes the oblate spheroidal angular function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns s

Value of the function

sp

Value of the derivative vs x

`scipy.special.obl_rad1(m, n, c, x) = <ufunc 'obl_rad1'>`

Oblate spheroidal radial function of the first kind and its derivative

Computes the oblate spheroidal radial function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns s

Value of the function

sp

Value of the derivative vs x

`scipy.special.obl_rad2(m, n, c, x) = <ufunc 'obl_rad2'>`

Oblate spheroidal radial function of the second kind and its derivative.

Computes the oblate spheroidal radial function of the second kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`.

Returns s

Value of the function

sp

Value of the derivative vs x

`scipy.special.pro_cv(m, n, c) = <ufunc 'pro_cv'>`

Characteristic value of prolate spheroidal function

Computes the characteristic value of prolate spheroidal wave functions of order *m*, *n* (n>=m) and spheroidal parameter *c*.

`scipy.special.obl_cv(m, n, c) = <ufunc 'obl_cv'>`

Characteristic value of oblate spheroidal function

Computes the characteristic value of oblate spheroidal wave functions of order *m*, *n* (n>=m) and spheroidal parameter *c*.

`scipy.special.pro_cv_seq(m, n, c)`

Characteristic values for prolate spheroidal wave functions.

Compute a sequence of characteristic values for the prolate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.

References

[R507]

`scipy.special.obl_cv_seq(m, n, c)`

Characteristic values for oblate spheroidal wave functions.

Compute a sequence of characteristic values for the oblate spheroidal wave functions for mode m and n'=m..n and spheroidal parameter c.

References

[R503]

The following functions require pre-computed characteristic value:

pro_ang1_cv(m, n, c, cv, x)	Prolate spheroidal angular function pro_ang1 for precomputed characteristic value
pro_rad1_cv(m, n, c, cv, x)	Prolate spheroidal radial function pro_rad1 for precomputed characteristic value
pro_rad2_cv(m, n, c, cv, x)	Prolate spheroidal radial function pro_rad2 for precomputed characteristic value
obl_ang1_cv(m, n, c, cv, x)	Oblate spheroidal angular function obl_ang1 for precomputed characteristic value
obl_rad1_cv(m, n, c, cv, x)	Oblate spheroidal radial function obl_rad1 for precomputed characteristic value
obl_rad2_cv(m, n, c, cv, x)	Oblate spheroidal radial function obl_rad2 for precomputed characteristic value

scipy.special.**pro_ang1_cv**(*m*, *n*, *c*, *cv*, *x*) **= <ufunc 'pro_ang1_cv'>**

Prolate spheroidal angular function pro_ang1 for precomputed characteristic value

Computes the prolate spheroidal angular function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s

Value of the function

sp

Value of the derivative vs x

scipy.special.**pro_rad1_cv**(*m*, *n*, *c*, *cv*, *x*) **= <ufunc 'pro_rad1_cv'>**

Prolate spheroidal radial function pro_rad1 for precomputed characteristic value

Computes the prolate spheroidal radial function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s

Value of the function

sp

Value of the derivative vs x

scipy.special.**pro_rad2_cv**(*m*, *n*, *c*, *cv*, *x*) **= <ufunc 'pro_rad2_cv'>**

Prolate spheroidal radial function pro_rad2 for precomputed characteristic value

Computes the prolate spheroidal radial function of the second kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s

Value of the function

sp

Value of the derivative vs x

scipy.special.**obl_ang1_cv**(*m*, *n*, *c*, *cv*, *x*) **= <ufunc 'obl_ang1_cv'>**

Oblate spheroidal angular function obl_ang1 for precomputed characteristic value

Computes the oblate spheroidal angular function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s

Value of the function

sp
Value of the derivative vs x

`scipy.special.`**`obl_rad1_cv`**(*m, n, c, cv, x*) **= <ufunc 'obl_rad1_cv'>**
Oblate spheroidal radial function obl_rad1 for precomputed characteristic value

Computes the oblate spheroidal radial function of the first kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s
Value of the function
sp
Value of the derivative vs x

`scipy.special.`**`obl_rad2_cv`**(*m, n, c, cv, x*) **= <ufunc 'obl_rad2_cv'>**
Oblate spheroidal radial function obl_rad2 for precomputed characteristic value

Computes the oblate spheroidal radial function of the second kind and its derivative (with respect to *x*) for mode parameters m>=0 and n>=m, spheroidal parameter *c* and `|x| < 1.0`. Requires pre-computed characteristic value.

Returns s
Value of the function
sp
Value of the derivative vs x

Kelvin Functions

kelvin(x)	Kelvin functions as complex numbers
kelvin_zeros(nt)	Compute nt zeros of all Kelvin functions.
ber(x)	Kelvin function ber.
bei(x)	Kelvin function bei
berp(x)	Derivative of the Kelvin function *ber*
beip(x)	Derivative of the Kelvin function *bei*
ker(x)	Kelvin function ker
kei(x)	Kelvin function ker
kerp(x)	Derivative of the Kelvin function ker
keip(x)	Derivative of the Kelvin function kei

`scipy.special.`**`kelvin`**(*x*) **= <ufunc 'kelvin'>**
Kelvin functions as complex numbers

Returns Be, Ke, Bep, Kep
The tuple (Be, Ke, Bep, Kep) contains complex numbers representing the real and imaginary Kelvin functions and their derivatives evaluated at *x*. For example, kelvin(x)[0].real = ber x and kelvin(x)[0].imag = bei x with similar relationships for ker and kei.

`scipy.special.`**`kelvin_zeros`**(*nt*)
Compute nt zeros of all Kelvin functions.

Returned in a length-8 tuple of arrays of length nt. The tuple contains the arrays of zeros of (ber, bei, ker, kei, ber', bei', ker', kei').

References

[R468]

scipy.special.**ber**(*x*) = <ufunc 'ber'>
Kelvin function ber.

scipy.special.**bei**(*x*) = <ufunc 'bei'>
Kelvin function bei

scipy.special.**berp**(*x*) = <ufunc 'berp'>
Derivative of the Kelvin function *ber*

scipy.special.**beip**(*x*) = <ufunc 'beip'>
Derivative of the Kelvin function *bei*

scipy.special.**ker**(*x*) = <ufunc 'ker'>
Kelvin function ker

scipy.special.**kei**(*x*) = <ufunc 'kei'>
Kelvin function ker

scipy.special.**kerp**(*x*) = <ufunc 'kerp'>
Derivative of the Kelvin function ker

scipy.special.**keip**(*x*) = <ufunc 'keip'>
Derivative of the Kelvin function kei

These are not universal functions:

ber_zeros(nt)	Compute nt zeros of the Kelvin function ber(x).
bei_zeros(nt)	Compute nt zeros of the Kelvin function bei(x).
berp_zeros(nt)	Compute nt zeros of the Kelvin function ber'(x).
beip_zeros(nt)	Compute nt zeros of the Kelvin function bei'(x).
ker_zeros(nt)	Compute nt zeros of the Kelvin function ker(x).
kei_zeros(nt)	Compute nt zeros of the Kelvin function kei(x).
kerp_zeros(nt)	Compute nt zeros of the Kelvin function ker'(x).
keip_zeros(nt)	Compute nt zeros of the Kelvin function kei'(x).

scipy.special.**ber_zeros**(*nt*)
Compute nt zeros of the Kelvin function ber(x).

References

[R380]

scipy.special.**bei_zeros**(*nt*)
Compute nt zeros of the Kelvin function bei(x).

References

[R378]

scipy.special.**berp_zeros**(*nt*)
Compute nt zeros of the Kelvin function ber'(x).

References

[R382]

scipy.special.**beip_zeros**(*nt*)
Compute nt zeros of the Kelvin function bei'(x).

References

[R379]

`scipy.special.`**`ker_zeros`**(*nt*)
Compute nt zeros of the Kelvin function ker(x).

References

[R469]

`scipy.special.`**`kei_zeros`**(*nt*)
Compute nt zeros of the Kelvin function kei(x).

`scipy.special.`**`kerp_zeros`**(*nt*)
Compute nt zeros of the Kelvin function ker'(x).

References

[R470]

`scipy.special.`**`keip_zeros`**(*nt*)
Compute nt zeros of the Kelvin function kei'(x).

References

[R467]

Combinatorics

comb(N, k[, exact, repetition])	The number of combinations of N things taken k at a time.
perm(N, k[, exact])	Permutations of N things taken k at a time, i.e., k-permutations of N.

`scipy.special.`**`comb`**(*N*, *k*, *exact=False*, *repetition=False*)
The number of combinations of N things taken k at a time.

This is often expressed as "N choose k".

Parameters

N : int, ndarray
Number of things.

k : int, ndarray
Number of elements taken.

exact : bool, optional
If *exact* is False, then floating point precision is used, otherwise exact long integer is computed.

repetition : bool, optional
If *repetition* is True, then the number of combinations with repetition is computed.

Returns

val : int, ndarray
The total number of combinations.

See also:

`binom` Binomial coefficient ufunc

Notes

- Array arguments accepted only for exact=False case.

•If k > N, N < 0, or k < 0, then a 0 is returned.

Examples

```
>>> from scipy.special import comb
>>> k = np.array([3, 4])
>>> n = np.array([10, 10])
>>> comb(n, k, exact=False)
array([ 120.,  210.])
>>> comb(10, 3, exact=True)
120L
>>> comb(10, 3, exact=True, repetition=True)
220L
```

`scipy.special.perm`(*N*, *k*, *exact=False*)

Permutations of N things taken k at a time, i.e., k-permutations of N.

It's also known as "partial permutations".

Parameters

N : int, ndarray
 Number of things.

k : int, ndarray
 Number of elements taken.

exact : bool, optional
 If *exact* is False, then floating point precision is used, otherwise exact long integer is computed.

Returns

val : int, ndarray
 The number of k-permutations of N.

Notes

•Array arguments accepted only for exact=False case.
•If k > N, N < 0, or k < 0, then a 0 is returned.

Examples

```
>>> from scipy.special import perm
>>> k = np.array([3, 4])
>>> n = np.array([10, 10])
>>> perm(n, k)
array([  720.,  5040.])
>>> perm(10, 3, exact=True)
720
```

Lambert W and Related Functions

`lambertw`(z[, k, tol])	Lambert W function.
`wrightomega`(z[, out])	Wright Omega function.

Other Special Functions

agm(a, b)	Arithmetic, Geometric Mean.
bernoulli(n)	Bernoulli numbers B0..Bn (inclusive).
binom(n, k)	Binomial coefficient

Continued on next page

Table 5.265 – continued from previous page

diric(x, n)	Periodic sinc function, also called the Dirichlet function.
euler(n)	Euler numbers E0..En (inclusive).
expn(n, x)	Exponential integral E_n
exp1(z)	Exponential integral E_1 of complex argument z
expi(x)	Exponential integral Ei
factorial(n[, exact])	The factorial of a number or array of numbers.
factorial2(n[, exact])	Double factorial.
factorialk(n, k[, exact])	Multifactorial of n of order k, n(!!...!).
shichi(x[, out])	Hyperbolic sine and cosine integrals.
sici(x[, out])	Sine and cosine integrals.
spence(z[, out])	Spence's function, also known as the dilogarithm.
zeta(x[, q, out])	Riemann zeta function.
zetac(x)	Riemann zeta function minus 1.

scipy.special.**agm**(*a*, *b*)

Arithmetic, Geometric Mean.

Start with a_0=a and b_0=b and iteratively compute

a_{n+1} = (a_n+b_n)/2 b_{n+1} = sqrt(a_n*b_n)

until a_n=b_n. The result is agm(a, b)

agm(a, b)=agm(b, a) agm(a, a) = a min(a, b) < agm(a, b) < max(a, b)

scipy.special.**bernoulli**(*n*)

Bernoulli numbers B0..Bn (inclusive).

References

[R381]

scipy.special.**binom**(*n*, *k*) **= <ufunc 'binom'>**

Binomial coefficient

See also:

comb The number of combinations of N things taken k at a time.

scipy.special.**diric**(*x*, *n*)

Periodic sinc function, also called the Dirichlet function.

The Dirichlet function is defined as:

```
diric(x) = sin(x * n/2) / (n * sin(x / 2)),
```

where *n* is a positive integer.

Parameters **x** : array_like

Input data

n : int

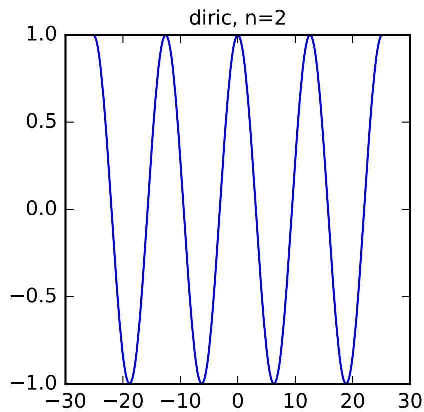
Integer defining the periodicity.

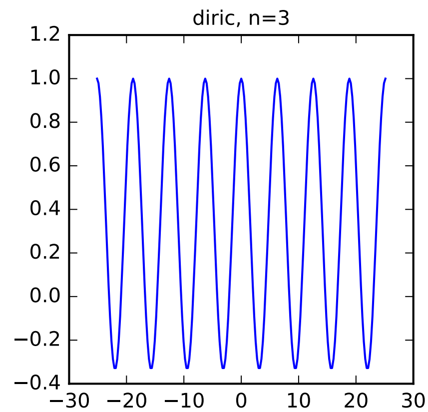
Returns **diric** : ndarray

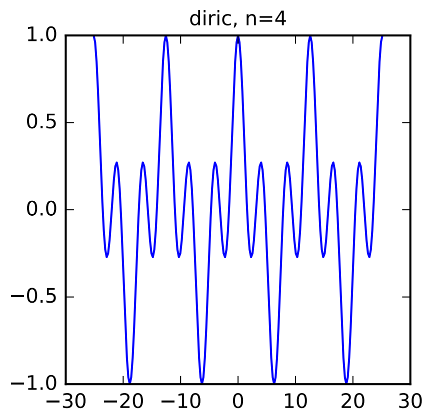
Examples

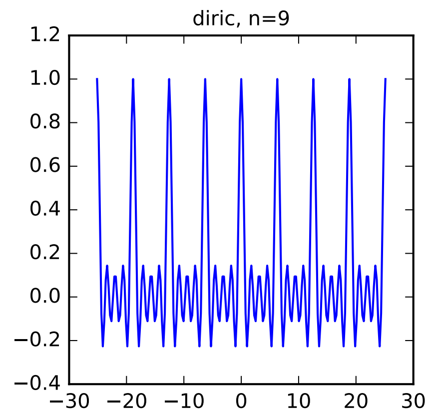
```
>>> from scipy import special
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-8*np.pi, 8*np.pi, num=201)
>>> plt.figure(figsize=(8, 8));
>>> for idx, n in enumerate([2, 3, 4, 9]):
...     plt.subplot(2, 2, idx+1)
...     plt.plot(x, special.diric(x, n))
...     plt.title('diric, n={}'.format(n))
>>> plt.show()
```









The following example demonstrates that `diric` gives the magnitudes (modulo the sign and scaling) of the Fourier coefficients of a rectangular pulse.

Suppress output of values that are effectively 0:

```
>>> np.set_printoptions(suppress=True)
```

Create a signal *x* of length *m* with *k* ones:

```
>>> m = 8
>>> k = 3
>>> x = np.zeros(m)
>>> x[:k] = 1
```

Use the FFT to compute the Fourier transform of *x*, and inspect the magnitudes of the coefficients:

```
>>> np.abs(np.fft.fft(x))
array([ 3.        ,  2.41421356,  1.        ,  0.41421356,  1.        ,
        0.41421356,  1.        ,  2.41421356])
```

Now find the same values (up to sign) using *diric*. We multiply by *k* to account for the different scaling conventions of `numpy.fft.fft` and *diric*:

```
>>> theta = np.linspace(0, 2*np.pi, m, endpoint=False)
>>> k * special.diric(theta, k)
array([ 3.        ,  2.41421356,  1.        , -0.41421356, -1.        ,
       -0.41421356,  1.        ,  2.41421356])
```

`scipy.special.`**`euler`**`(`*n*`)`
Euler numbers E0..En (inclusive).

References

[R414]

`scipy.special.`**`expn`**`(`*n, x*`)` **= <ufunc 'expn'>**
Exponential integral E_n

Returns the exponential integral for integer *n* and non-negative *x* and *n*:

```
integral(exp(-x*t) / t**n, t=1..inf).
```

`scipy.special.`**`exp1`**`(`*z*`)` **= <ufunc 'exp1'>**
Exponential integral E_1 of complex argument z

```
integral(exp(-z*t)/t, t=1..inf).
```

`scipy.special.`**`expi`**`(`*x*`)` **= <ufunc 'expi'>**
Exponential integral Ei

Defined as:

```
integral(exp(t)/t, t=-inf..x)
```

See *expn* for a different exponential integral.

`scipy.special.`**`factorial`**`(`*n, exact=False*`)`
The factorial of a number or array of numbers.

The factorial of non-negative integer *n* is the product of all positive integers less than or equal to *n*:

```
n! = n * (n - 1) * (n - 2) * ... * 1
```

Parameters **n** : int or array_like of ints
Input values. If `n < 0`, the return value is 0.
exact : bool, optional

If True, calculate the answer exactly using long integer arithmetic. If False, result is approximated in floating point rapidly using the *gamma* function. Default is False.

Returns **nf** : float or int or ndarray

Factorial of *n*, as integer or float depending on *exact*.

Notes

For arrays with `exact=True`, the factorial is computed only once, for the largest input, with each other result computed in the process. The output dtype is increased to `int64` or `object` if necessary.

With `exact=False` the factorial is approximated using the gamma function:

$$n! = \Gamma(n + 1)$$

Examples

```
>>> from scipy.special import factorial
>>> arr = np.array([3, 4, 5])
>>> factorial(arr, exact=False)
array([   6.,   24.,  120.])
>>> factorial(arr, exact=True)
array([  6,  24, 120])
>>> factorial(5, exact=True)
120L
```

scipy.special.**factorial2**(*n*, *exact=False*)

Double factorial.

This is the factorial with every second value skipped. E.g., `7!! = 7 * 5 * 3 * 1`. It can be approximated numerically as:

```
n!! = special.gamma(n/2+1)*2**((m+1)/2)/sqrt(pi)   n odd
    = 2**(n/2) * (n/2)!                            n even
```

Parameters **n** : int or array_like

Calculate `n!!`. Arrays are only supported with *exact* set to False. If `n < 0`, the return value is 0.

exact : bool, optional

The result can be approximated rapidly using the gamma-formula above (default). If *exact* is set to True, calculate the answer exactly using integer arithmetic.

Returns **nff** : float or int

Double factorial of *n*, as an int or a float depending on *exact*.

Examples

```
>>> from scipy.special import factorial2
>>> factorial2(7, exact=False)
array(105.00000000000001)
>>> factorial2(7, exact=True)
105L
```

scipy.special.**factorialk**(*n*, *k*, *exact=True*)

Multifactorial of n of order k, n(!!...!).

This is the multifactorial of n skipping k values. For example,

factorialk(17, 4) = 17!!!! = 17 * 13 * 9 * 5 * 1

In particular, for any integer n, we have

factorialk(n, 1) = factorial(n)
factorialk(n, 2) = factorial2(n)

Parameters **n** : int
Calculate multifactorial. If n < 0, the return value is 0.

k : int
Order of multifactorial.

exact : bool, optional
If exact is set to True, calculate the answer exactly using integer arithmetic.

Returns **val** : int
Multifactorial of n.

Raises **NotImplementedError**
Raises when exact is False

Examples

```
>>> from scipy.special import factorialk
>>> factorialk(5, 1, exact=True)
120L
>>> factorialk(5, 3, exact=True)
10L
```

scipy.special.**shichi**(*x*, *out=None*) = <ufunc 'shichi'>
Hyperbolic sine and cosine integrals.

The hyperbolic sine integral is

$$\int_0^x \frac{\sinh t}{t} dt$$

and the hyperbolic cosine integral is

$$\gamma + \log(x) + \int_0^x \frac{\cosh t - 1}{t} dt$$

where γ is Euler's constant and log is the principle branch of the logarithm.

Parameters **x** : array_like
Real or complex points at which to compute the hyperbolic sine and cosine integrals.

Returns **si** : ndarray
Hyperbolic sine integral at x

ci : ndarray
Hyperbolic cosine integral at x

Notes

For real arguments with `x < 0`, `chi` is the real part of the hyperbolic cosine integral. For such points `chi(x)` and `chi(x + 0j)` differ by a factor of `1j*pi`.

For real arguments the function is computed by calling Cephes' *[R514]* *shichi* routine. For complex arguments the algorithm is based on Mpmath's *[R515]* *shi* and *chi* routines.

References

[R514], *[R515]*

`scipy.special.sici(x, out=None) = <ufunc 'sici'>`

Sine and cosine integrals.

The sine integral is

$$\int_0^x \frac{\sin t}{t} dt$$

and the cosine integral is

$$\gamma + \log(x) + \int_0^x \frac{\cos t - 1}{t} dt$$

where γ is Euler's constant and $\log$ is the principle branch of the logarithm.

Parameters **x** : array_like

Real or complex points at which to compute the sine and cosine integrals.

Returns **si** : ndarray

Sine integral at `x`

ci : ndarray

Cosine integral at `x`

Notes

For real arguments with `x < 0`, `ci` is the real part of the cosine integral. For such points `ci(x)` and `ci(x + 0j)` differ by a factor of `1j*pi`.

For real arguments the function is computed by calling Cephes' *[R516] sici* routine. For complex arguments the algorithm is based on Mpmath's *[R517] si* and *ci* routines.

References

[R516], *[R517]*

`scipy.special.spence(z, out=None) = <ufunc 'spence'>`

Spence's function, also known as the dilogarithm.

It is defined to be

$$\int_0^z \frac{\log(t)}{1-t} dt$$

for complex z, where the contour of integration is taken to avoid the branch cut of the logarithm. Spence's function is analytic everywhere except the negative real axis where it has a branch cut.

Parameters **z** : array_like

Points at which to evaluate Spence's function

Returns **s** : ndarray

Computed values of Spence's function

Notes

There is a different convention which defines Spence's function by the integral

$$-\int_0^z \frac{\log(1-t)}{t} dt;$$

this is our `spence(1 - z)`.

`scipy.special.zeta(x, q=None, out=None)`

Riemann zeta function.

The two-argument version is the Hurwitz zeta function:

$$\zeta(x, q) = \sum_{k=0}^{\infty} \frac{1}{(k+q)^x},$$

Riemann zeta function corresponds to `q = 1`.

See also:

zetac

`scipy.special.`**`zetac`**`(x)` **= <ufunc 'zetac'>**

Riemann zeta function minus 1.

This function is defined as

$$\zeta(x) = \sum_{k=2}^{\infty} 1/k^x,$$

where `x > 1`.

See also:

zeta

Convenience Functions

cbrt(x)	Cube root of *x*
exp10(x)	10**x
exp2(x)	2**x
radian(d, m, s)	Convert from degrees to radians
cosdg(x)	Cosine of the angle *x* given in degrees.
sindg(x)	Sine of angle given in degrees
tandg(x)	Tangent of angle x given in degrees.
cotdg(x)	Cotangent of the angle *x* given in degrees.
log1p(x)	Calculates log(1+x) for use when *x* is near zero
expm1(x)	exp(x) - 1 for use when *x* is near zero.
cosm1(x)	cos(x) - 1 for use when *x* is near zero.
round(x)	Round to nearest integer
xlogy(x, y)	Compute `x*log(y)` so that the result is 0 if `x = 0`.
xlog1py(x, y)	Compute `x*log1p(y)` so that the result is 0 if `x = 0`.
logsumexp(a[, axis, b, keepdims, return_sign])	Compute the log of the sum of exponentials of input elements.
exprel(x)	Relative error exponential, (exp(x)-1)/x, for use when *x* is near zero.
sinc(x)	Return the sinc function.

`scipy.special.`**`cbrt`**`(x)` **= <ufunc 'cbrt'>**

Cube root of *x*

`scipy.special.`**`exp10`**`(x)` **= <ufunc 'exp10'>**

10**x

`scipy.special.`**`exp2`**`(x)` **= <ufunc 'exp2'>**

2**x

`scipy.special.`**`radian`**(*d*, *m*, *s*) = <ufunc 'radian'>
Convert from degrees to radians

Returns the angle given in (d)egrees, (m)inutes, and (s)econds in radians.

`scipy.special.`**`cosdg`**(*x*) = <ufunc 'cosdg'>
Cosine of the angle *x* given in degrees.

`scipy.special.`**`sindg`**(*x*) = <ufunc 'sindg'>
Sine of angle given in degrees

`scipy.special.`**`tandg`**(*x*) = <ufunc 'tandg'>
Tangent of angle x given in degrees.

`scipy.special.`**`cotdg`**(*x*) = <ufunc 'cotdg'>
Cotangent of the angle *x* given in degrees.

`scipy.special.`**`log1p`**(*x*) = <ufunc 'log1p'>
Calculates log(1+x) for use when *x* is near zero

`scipy.special.`**`expm1`**(*x*) = <ufunc 'expm1'>
exp(x) - 1 for use when *x* is near zero.

`scipy.special.`**`cosm1`**(*x*) = <ufunc 'cosm1'>
cos(x) - 1 for use when *x* is near zero.

`scipy.special.`**`round`**(*x*) = <ufunc 'round'>
Round to nearest integer

Returns the nearest integer to *x* as a double precision floating point result. If *x* ends in 0.5 exactly, the nearest even integer is chosen.

`scipy.special.`**`xlogy`**(*x*, *y*) = <ufunc 'xlogy'>
Compute `x*log(y)` so that the result is 0 if `x = 0`.

Parameters **x** : array_like
Multiplier
y : array_like
Argument

Returns **z** : array_like
Computed x*log(y)

Notes

New in version 0.13.0.

`scipy.special.`**`xlog1py`**(*x*, *y*) = <ufunc 'xlog1py'>
Compute `x*log1p(y)` so that the result is 0 if `x = 0`.

Parameters **x** : array_like
Multiplier
y : array_like
Argument

Returns **z** : array_like
Computed x*log1p(y)

Notes

New in version 0.13.0.

`scipy.special.`**`logsumexp`**(*a*, *axis=None*, *b=None*, *keepdims=False*, *return_sign=False*)
Compute the log of the sum of exponentials of input elements.

Parameters **a** : array_like

Input array.

axis : None or int or tuple of ints, optional

Axis or axes over which the sum is taken. By default *axis* is None, and all elements are summed.

New in version 0.11.0.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array.

New in version 0.15.0.

b : array-like, optional

Scaling factor for exp(*a*) must be of the same shape as *a* or broadcastable to *a*. These values may be negative in order to implement subtraction.

New in version 0.12.0.

return_sign : bool, optional

If this is set to True, the result will be a pair containing sign information; if False, results that are negative will be returned as NaN. Default is False (no sign information).

New in version 0.16.0.

Returns

——-

res : ndarray

The result, `np.log(np.sum(np.exp(a)))` calculated in a numerically more stable way. If *b* is given then `np.log(np.sum(b*np.exp(a)))` is returned.

sgn : ndarray

If return_sign is True, this will be an array of floating-point numbers matching res and +1, 0, or -1 depending on the sign of the result. If False, only one result is returned.

See also:

`numpy.logaddexp`, `numpy.logaddexp2`

Notes

Numpy has a logaddexp function which is very similar to `logsumexp`, but only handles two arguments. *logaddexp.reduce* is similar to this function, but may be less stable.

Examples

```
>>> from scipy.special import logsumexp
>>> a = np.arange(10)
>>> np.log(np.sum(np.exp(a)))
9.4586297444267107
>>> logsumexp(a)
9.4586297444267107
```

With weights

```
>>> a = np.arange(10)
>>> b = np.arange(10, 0, -1)
>>> logsumexp(a, b=b)
9.9170178533034665
>>> np.log(np.sum(b*np.exp(a)))
9.9170178533034647
```

Returning a sign flag

```
>>> logsumexp([1,2],b=[1,-1],return_sign=True)
(1.5413248546129181, -1.0)
```

Notice that *logsumexp* does not directly support masked arrays. To use it on a masked array, convert the mask into zero weights:

```
>>> a = np.ma.array([np.log(2), 2, np.log(3)],
...                  mask=[False, True, False])
>>> b = (~a.mask).astype(int)
>>> logsumexp(a.data, b=b), np.log(5)
1.6094379124341005, 1.6094379124341005
```

`scipy.special.`**`exprel`**(*x*) = **<ufunc 'exprel'>**

Relative error exponential, (exp(x)-1)/x, for use when *x* is near zero.

Parameters **x** : ndarray

Input array.

Returns **res** : ndarray

Output array.

See also:

expm1,

`scipy.special.`**`sinc`**(*x*)

Return the sinc function.

The sinc function is $\sin(\pi x)/(\pi x)$.

Parameters **x** : ndarray

Array (possibly multi-dimensional) of values for which to to calculate `sinc(x)`.

Returns **out** : ndarray

`sinc(x)`, which has the same shape as the input.

Notes

`sinc(0)` is the limit value 1.

The name sinc is short for "sine cardinal" or "sinus cardinalis".

The sinc function is used in various signal processing applications, including in anti-aliasing, in the construction of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the sinc function.

References

[R518], *[R519]*

Examples

```
>>> x = np.linspace(-4, 4, 41)
>>> np.sinc(x)
array([ -3.89804309e-17,  -4.92362781e-02,  -8.40918587e-02,
        -8.90384387e-02,  -5.84680802e-02,   3.89804309e-17,
         6.68206631e-02,   1.16434881e-01,   1.26137788e-01,
         8.50444803e-02,  -3.89804309e-17,  -1.03943254e-01,
        -1.89206682e-01,  -2.16236208e-01,  -1.55914881e-01,
```

```
         3.89804309e-17,   2.33872321e-01,   5.04551152e-01,
         7.56826729e-01,   9.35489284e-01,   1.00000000e+00,
         9.35489284e-01,   7.56826729e-01,   5.04551152e-01,
         2.33872321e-01,   3.89804309e-17,  -1.55914881e-01,
        -2.16236208e-01,  -1.89206682e-01,  -1.03943254e-01,
        -3.89804309e-17,   8.50444803e-02,   1.26137788e-01,
         1.16434881e-01,   6.68206631e-02,   3.89804309e-17,
        -5.84680802e-02,  -8.90384387e-02,  -8.40918587e-02,
        -4.92362781e-02,  -3.89804309e-17])
```

```
>>> plt.plot(x, np.sinc(x))
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Sinc Function")
<matplotlib.text.Text object at 0x...>
>>> plt.ylabel("Amplitude")
<matplotlib.text.Text object at 0x...>
>>> plt.xlabel("X")
<matplotlib.text.Text object at 0x...>
>>> plt.show()
```

It works in 2-D as well:

```
>>> x = np.linspace(-4, 4, 401)
>>> xx = np.outer(x, x)
>>> plt.imshow(np.sinc(xx))
<matplotlib.image.AxesImage object at 0x...>
```

5.27 Statistical functions (`scipy.stats`)

This module contains a large number of probability distributions as well as a growing library of statistical functions.

Each univariate distribution is an instance of a subclass of *rv_continuous* (*rv_discrete* for discrete distributions):

rv_continuous([momtype, a, b, xtol, ...])	A generic continuous random variable class meant for subclassing.
rv_discrete([a, b, name, badvalue, ...])	A generic discrete random variable class meant for subclassing.
rv_histogram(histogram, *args, **kwargs)	Generates a distribution given by a histogram.

class `scipy.stats.`**`rv_continuous`**(*momtype=1, a=None, b=None, xtol=1e-14, badvalue=None, name=None, longname=None, shapes=None, extradoc=None, seed=None*)

A generic continuous random variable class meant for subclassing.

rv_continuous is a base class to construct specific distribution classes and instances for continuous random variables. It cannot be used directly as a distribution.

Parameters

momtype : int, optional
: The type of generic moment calculation to use: 0 for pdf, 1 (default) for ppf.

a : float, optional
: Lower bound of the support of the distribution, default is minus infinity.

b : float, optional

Upper bound of the support of the distribution, default is plus infinity.

xtol : float, optional
The tolerance for fixed point calculation for generic ppf.

badvalue : float, optional
The value in a result arrays that indicates a value that for which some argument restriction is violated, default is np.nan.

name : str, optional
The name of the instance. This string is used to construct the default example for distributions.

longname : str, optional
This string is used as part of the first line of the docstring returned when a subclass has no docstring of its own. Note: *longname* exists for backwards compatibility, do not use for new subclasses.

shapes : str, optional
The shape of the distribution. For example `"m, n"` for a distribution that takes two integers as the two shape arguments for all its methods. If not provided, shape parameters will be inferred from the signature of the private methods, `_pdf` and `_cdf` of the instance.

extradoc : str, optional, deprecated
This string is used as the last part of the docstring returned when a subclass has no docstring of its own. Note: *extradoc* exists for backwards compatibility, do not use for new subclasses.

seed : None or int or `numpy.random.RandomState` instance, optional
This parameter defines the RandomState object to use for drawing random variates. If None (or np.random), the global np.random state is used. If integer, it is used to seed the local RandomState instance. Default is None.

Notes

Public methods of an instance of a distribution class (e.g., `pdf`, `cdf`) check their arguments and pass valid arguments to private, computational methods (`_pdf`, `_cdf`). For `pdf(x)`, `x` is valid if it is within the support of a distribution, `self.a <= x <= self.b`. Whether a shape parameter is valid is decided by an `_argcheck` method (which defaults to checking that its arguments are strictly positive.)

Subclassing

New random variables can be defined by subclassing the *rv_continuous* class and re-defining at least the `_pdf` or the `_cdf` method (normalized to location 0 and scale 1).

If positive argument checking is not correct for your RV then you will also need to re-define the `_argcheck` method.

Correct, but potentially slow defaults exist for the remaining methods but for speed and/or accuracy you can over-ride:

```
_logpdf, _cdf, _logcdf, _ppf, _rvs, _isf, _sf, _logsf
```

Rarely would you override `_isf`, `_sf` or `_logsf`, but you could.

Methods that can be overwritten by subclasses

```
_rvs
_pdf
_cdf
_sf
_ppf
_isf
_stats
```

```
_munp
_entropy
_argcheck
```

There are additional (internal and private) generic methods that can be useful for cross-checking and for debugging, but might work in all cases when directly called.

A note on `shapes`: subclasses need not specify them explicitly. In this case, *shapes* will be automatically deduced from the signatures of the overridden methods (`pdf`, `cdf` etc). If, for some reason, you prefer to avoid relying on introspection, you can specify `shapes` explicitly as an argument to the instance constructor.

Frozen Distributions

Normally, you must provide shape parameters (and, optionally, location and scale parameters to each call of a method of a distribution.

Alternatively, the object may be called (as a function) to fix the shape, location, and scale parameters returning a "frozen" continuous RV object:

rv = generic(<shape(s)>, loc=0, scale=1)
: frozen RV object with the same methods but holding the given shape, location, and scale fixed

Statistics

Statistics are computed using numerical integration by default. For speed you can redefine this using `_stats`:

- take shape parameters and return mu, mu2, g1, g2
- If you can't compute one of these, return it as None
- Can also be defined with a keyword argument `moments`, which is a string composed of "m", "v", "s", and/or "k". Only the components appearing in string should be computed and returned in the order "m", "v", "s", or "k" with missing values returned as None.

Alternatively, you can override `_munp`, which takes `n` and shape parameters and returns the n-th non-central moment of the distribution.

Examples

To create a new Gaussian distribution, we would do the following:

```
>>> from scipy.stats import rv_continuous
>>> class gaussian_gen(rv_continuous):
...     "Gaussian distribution"
...     def _pdf(self, x):
...         return np.exp(-x**2 / 2.) / np.sqrt(2.0 * np.pi)
>>> gaussian = gaussian_gen(name='gaussian')
```

`scipy.stats` distributions are *instances*, so here we subclass `rv_continuous` and create an instance. With this, we now have a fully functional distribution with all relevant methods automagically generated by the framework.

Note that above we defined a standard normal distribution, with zero mean and unit variance. Shifting and scaling of the distribution can be done by using `loc` and `scale` parameters: `gaussian.pdf(x, loc, scale)` essentially computes `y = (x - loc) / scale` and `gaussian._pdf(y) / scale`.

Attributes

random_state	Get or set the RandomState object for generating random variates.

`rv_continuous.`**`random_state`**
: Get or set the RandomState object for generating random variates.

This can be either None or an existing RandomState object.

If None (or np.random), use the RandomState singleton used by np.random. If already a RandomState instance, use it. If an int, use a new RandomState instance seeded with seed.

Methods

`rvs`(*args, **kwds)	Random variates of given type.
`pdf`(x, *args, **kwds)	Probability density function at x of the given RV.
`logpdf`(x, *args, **kwds)	Log of the probability density function at x of the given RV.
`cdf`(x, *args, **kwds)	Cumulative distribution function of the given RV.
`logcdf`(x, *args, **kwds)	Log of the cumulative distribution function at x of the given RV.
`sf`(x, *args, **kwds)	Survival function (1 - `cdf`) at x of the given RV.
`logsf`(x, *args, **kwds)	Log of the survival function of the given RV.
`ppf`(q, *args, **kwds)	Percent point function (inverse of `cdf`) at q of the given RV.
`isf`(q, *args, **kwds)	Inverse survival function (inverse of `sf`) at q of the given RV.
`moment`(n, *args, **kwds)	n-th order non-central moment of distribution.
`stats`(*args, **kwds)	Some statistics of the given RV.
`entropy`(*args, **kwds)	Differential entropy of the RV.
`expect`([func, args, loc, scale, lb, ub, ...])	Calculate expected value of a function with respect to the distribution.
`median`(*args, **kwds)	Median of the distribution.
`mean`(*args, **kwds)	Mean of the distribution.
`std`(*args, **kwds)	Standard deviation of the distribution.
`var`(*args, **kwds)	Variance of the distribution.
`interval`(alpha, *args, **kwds)	Confidence interval with equal areas around the median.
`__call__`(*args, **kwds)	Freeze the distribution for the given arguments.
`fit`(data, *args, **kwds)	Return MLEs for shape (if applicable), location, and scale parameters from data.
`fit_loc_scale`(data, *args)	Estimate loc and scale parameters from data using 1st and 2nd moments.
`nnlf`(theta, x)	Return negative loglikelihood function.

`rv_continuous.`**`rvs`**(**args*, ***kwds*)

Random variates of given type.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional

Scale parameter (default=1).

size : int or tuple of ints, optional

Defining number of random variates (default is 1).

random_state : None or int or `np.random.RandomState` instance, optional

If int or RandomState, use it for drawing the random variates. If None, rely on `self.random_state`. Default is None.

Returns **rvs** : ndarray or scalar

Random variates of given *size*.

`rv_continuous.`**`pdf`**(*x*, **args*, ***kwds*)

Probability density function at x of the given RV.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
pdf : ndarray
Probability density function evaluated at x

`rv_continuous.`**`logpdf`**(*x*, **args*, ***kwds*)

Log of the probability density function at x of the given RV.

This uses a more numerically accurate calculation if available.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
logpdf : array_like
Log of the probability density function evaluated at x

`rv_continuous.`**`cdf`**(*x*, **args*, ***kwds*)

Cumulative distribution function of the given RV.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
cdf : ndarray
Cumulative distribution function evaluated at *x*

`rv_continuous.`**`logcdf`**(*x*, **args*, ***kwds*)

Log of the cumulative distribution function at x of the given RV.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
logcdf : array_like

Log of the cumulative distribution function evaluated at x

`rv_continuous.sf(x, *args, **kwds)`

Survival function (1 - `cdf`) at x of the given RV.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
sf : array_like
Survival function evaluated at x

`rv_continuous.logsf(x, *args, **kwds)`

Log of the survival function of the given RV.

Returns the log of the "survival function," defined as (1 - `cdf`), evaluated at *x*.

Parameters
x : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
logsf : ndarray
Log of the survival function evaluated at *x*.

`rv_continuous.ppf(q, *args, **kwds)`

Percent point function (inverse of `cdf`) at q of the given RV.

Parameters
q : array_like
lower tail probability
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns
x : array_like
quantile corresponding to the lower tail probability q.

`rv_continuous.isf(q, *args, **kwds)`

Inverse survival function (inverse of `sf`) at q of the given RV.

Parameters
q : array_like
upper tail probability
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional

scale parameter (default=1)

Returns **x** : ndarray or scalar

Quantile corresponding to the upper tail probability q.

`rv_continuous.moment(n, *args, **kwds)`

n-th order non-central moment of distribution.

Parameters **n** : int, n >= 1

Order of moment.

arg1, arg2, arg3,... : float

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

`rv_continuous.stats(*args, **kwds)`

Some statistics of the given RV.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional (continuous RVs only)

scale parameter (default=1)

moments : str, optional

composed of letters ['mvsk'] defining which moments to compute: 'm' = mean, 'v' = variance, 's' = (Fisher's) skew, 'k' = (Fisher's) kurtosis. (default is 'mv')

Returns **stats** : sequence

of requested moments.

`rv_continuous.entropy(*args, **kwds)`

Differential entropy of the RV.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional (continuous distributions only).

Scale parameter (default=1).

Notes

Entropy is defined base *e*:

```
>>> drv = rv_discrete(values=((0, 1), (0.5, 0.5)))
>>> np.allclose(drv.entropy(), np.log(2.0))
True
```

`rv_continuous.expect(func=None, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`

Calculate expected value of a function with respect to the distribution.

The expected value of a function `f(x)` with respect to a distribution `dist` is defined as:

```
        ubound
E[x] = Integral(f(x) * dist.pdf(x))
        lbound
```

Parameters **func** : callable, optional

Function for which integral is calculated. Takes only one argument. The default is the identity mapping f(x) = x.

args : tuple, optional

Shape parameters of the distribution.

loc : float, optional

Location parameter (default=0).

scale : float, optional

Scale parameter (default=1).

lb, ub : scalar, optional

Lower and upper bound for integration. Default is set to the support of the distribution.

conditional : bool, optional

If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.

Additional keyword arguments are passed to the integration routine.

Returns **expect** : float

The calculated expected value.

Notes

The integration behavior of this function is inherited from *integrate.quad*.

`rv_continuous.`**`median`**`(`**args*, ***kwds*`)`

Median of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

Location parameter, Default is 0.

scale : array_like, optional

Scale parameter, Default is 1.

Returns **median** : float

The median of the distribution.

See also:

`stats.distributions.rv_discrete.ppf`

Inverse of the CDF

`rv_continuous.`**`mean`**`(`**args*, ***kwds*`)`

Mean of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **mean** : float

the mean of the distribution

`rv_continuous.`**`std`**`(`**args*, ***kwds*`)`

Standard deviation of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **std** : float
standard deviation of the distribution

`rv_continuous.`**`var`**`(`**args*, ***kwds*`)`

Variance of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **var** : float
the variance of the distribution

`rv_continuous.`**`interval`**`(`*alpha*, **args*, ***kwds*`)`

Confidence interval with equal areas around the median.

Parameters **alpha** : array_like of float
Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].
arg1, arg2, ... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
location parameter, Default is 0.
scale : array_like, optional
scale parameter, Default is 1.

Returns **a, b** : ndarray of float
end-points of range that contain `100 * alpha %` of the rv's possible values.

`rv_continuous.`**`__call__`**`(`**args*, ***kwds*`)`

Freeze the distribution for the given arguments.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

Returns **rv_frozen** : rv_frozen instance
The frozen distribution.

`rv_continuous.`**`fit`**`(`*data*, **args*, ***kwds*`)`

Return MLEs for shape (if applicable), location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, `self._fitstart(data)` is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments `f0`, `f1`, ..., `fn` (for shape parameters) and `floc` and `fscale` (for location and scale parameters, respectively).

Parameters **data** : array_like

Data to use in calculating the MLEs.

args : floats, optional

Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to `_fitstart(data)`). No default value.

kwds : floats, optional

Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:

- f0...fn : hold respective shape parameters fixed. Alternatively, shape parameters to fix can be specified by name. For example, if `self.shapes == "a, b"`, ``fa``and ``fix_a`` are equivalent to `f0`, and `fb` and `fix_b` are equivalent to `f1`.
- floc : hold location parameter fixed to specified value.
- fscale : hold scale parameter fixed to specified value.
- optimizer : The optimizer to use. The optimizer must take `func`, and starting position as the first two arguments, plus `args` (for extra arguments to pass to the function to be optimized) and `disp=0` to suppress output as keyword arguments.

Returns **mle_tuple** : tuple of floats

MLEs for any shape parameters (if applicable), followed by those for location and scale. For most random variables, shape statistics will be returned, but there are exceptions (e.g. `norm`).

Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

Examples

Generate some data to fit: draw random variates from the *beta* distribution

```
>>> from scipy.stats import beta
>>> a, b = 1., 2.
>>> x = beta.rvs(a, b, size=1000)
```

Now we can fit all four parameters (`a`, `b`, `loc` and `scale`):

```
>>> a1, b1, loc1, scale1 = beta.fit(x)
```

We can also use some prior knowledge about the dataset: let's keep `loc` and `scale` fixed:

```
>>> a1, b1, loc1, scale1 = beta.fit(x, floc=0, fscale=1)
>>> loc1, scale1
(0, 1)
```

We can also keep shape parameters fixed by using `f`-keywords. To keep the zero-th shape parameter `a` equal 1, use `f0=1` or, equivalently, `fa=1`:

```
>>> a1, b1, loc1, scale1 = beta.fit(x, fa=1, floc=0, fscale=1)
>>> a1
1
```

Not all distributions return estimates for the shape parameters. `norm` for example just returns estimates for location and scale:

```
>>> from scipy.stats import norm
>>> x = norm.rvs(a, b, size=1000, random_state=123)
>>> loc1, scale1 = norm.fit(x)
>>> loc1, scale1
(0.92087172783841631, 2.0015750750324668)
```

`rv_continuous.`**`fit_loc_scale`**(*data*, **args*)

Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters **data** : array_like

Data to fit.

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns **Lhat** : float

Estimated location parameter for the data.

Shat : float

Estimated scale parameter for the data.

`rv_continuous.`**`nnlf`**(*theta*, *x*)

Return negative loglikelihood function.

Notes

This is `-sum(log pdf(x, theta), axis=0)` where *theta* are the parameters (including loc and scale).

class `scipy.stats.`**`rv_discrete`**(*a=0*, *b=inf*, *name=None*, *badvalue=None*, *moment_tol=1e-08*, *values=None*, *inc=1*, *longname=None*, *shapes=None*, *extradoc=None*, *seed=None*)

A generic discrete random variable class meant for subclassing.

`rv_discrete` is a base class to construct specific distribution classes and instances for discrete random variables. It can also be used to construct an arbitrary distribution defined by a list of support points and corresponding probabilities.

Parameters **a** : float, optional

Lower bound of the support of the distribution, default: 0

b : float, optional

Upper bound of the support of the distribution, default: plus infinity

moment_tol : float, optional

The tolerance for the generic calculation of moments.

values : tuple of two array_like, optional

`(xk, pk)` where `xk` are integers with non-zero probabilities `pk` with `sum(pk) = 1`.

inc : integer, optional

Increment for the support of the distribution. Default is 1. (other values have not been tested)

badvalue : float, optional

The value in a result arrays that indicates a value that for which some argument restriction is violated, default is np.nan.

name : str, optional

The name of the instance. This string is used to construct the default example for distributions.

longname : str, optional

This string is used as part of the first line of the docstring returned when a subclass has no docstring of its own. Note: *longname* exists for backwards compatibility, do not use for new subclasses.

shapes : str, optional

The shape of the distribution. For example "m, n" for a distribution that takes two integers as the two shape arguments for all its methods If not provided, shape parameters will be inferred from the signatures of the private methods, `_pmf` and `_cdf` of the instance.

extradoc : str, optional

This string is used as the last part of the docstring returned when a subclass has no docstring of its own. Note: *extradoc* exists for backwards compatibility, do not use for new subclasses.

seed : None or int or `numpy.random.RandomState` instance, optional

This parameter defines the RandomState object to use for drawing random variates. If None, the global np.random state is used. If integer, it is used to seed the local RandomState instance. Default is None.

Notes

This class is similar to *rv_continuous*, the main differences being:

- the support of the distribution is a set of integers
- instead of the probability density function, `pdf` (and the corresponding private `_pdf`), this class defines the *probability mass function*, *pmf* (and the corresponding private `_pmf`.)
- scale parameter is not defined.

To create a new discrete distribution, we would do the following:

```
>>> from scipy.stats import rv_discrete
>>> class poisson_gen(rv_discrete):
...     "Poisson distribution"
...     def _pmf(self, k, mu):
...         return exp(-mu) * mu**k / factorial(k)
```

and create an instance:

```
>>> poisson = poisson_gen(name="poisson")
```

Note that above we defined the Poisson distribution in the standard form. Shifting the distribution can be done by providing the `loc` parameter to the methods of the instance. For example, `poisson.pmf(x, mu, loc)` delegates the work to `poisson._pmf(x-loc, mu)`.

Discrete distributions from a list of probabilities

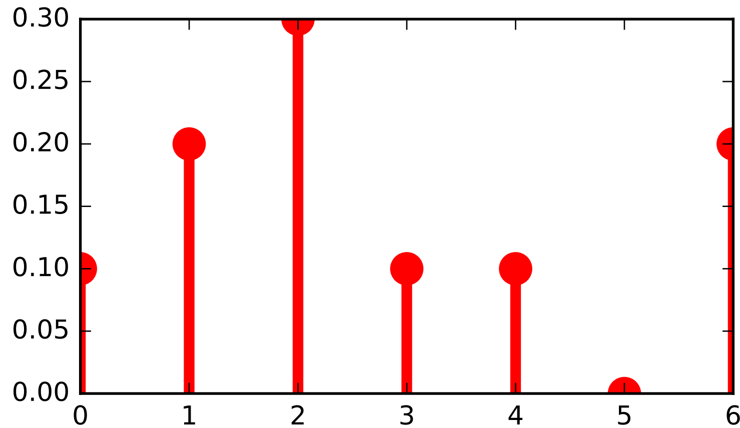
Alternatively, you can construct an arbitrary discrete rv defined on a finite set of values `xk` with `Prob{X=xk} = pk` by using the `values` keyword argument to the *rv_discrete* constructor.

Examples

Custom made discrete distribution:

```
>>> from scipy import stats
>>> xk = np.arange(7)
>>> pk = (0.1, 0.2, 0.3, 0.1, 0.1, 0.0, 0.2)
>>> custm = stats.rv_discrete(name='custm', values=(xk, pk))
>>>
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
>>> ax.plot(xk, custm.pmf(xk), 'ro', ms=12, mec='r')
```

```
>>> ax.vlines(xk, 0, custm.pmf(xk), colors='r', lw=4)
>>> plt.show()
```



Random number generation:

```
>>> R = custm.rvs(size=100)
```

Attributes

random_state	Get or set the RandomState object for generating random variates.
return_integers	*return_integers* is deprecated!

rv_discrete.**random_state**

Get or set the RandomState object for generating random variates.

This can be either None or an existing RandomState object.

If None (or np.random), use the RandomState singleton used by np.random. If already a RandomState instance, use it. If an int, use a new RandomState instance seeded with seed.

rv_discrete.**return_integers**

return_integers is deprecated! *return_integers* attribute is not used anywhere any longer and is deprecated in scipy 0.18.

Methods

rvs(*args, **kwargs)	Random variates of given type.
pmf(k, *args, **kwds)	Probability mass function at k of the given RV.
logpmf(k, *args, **kwds)	Log of the probability mass function at k of the given RV.
cdf(k, *args, **kwds)	Cumulative distribution function of the given RV.
logcdf(k, *args, **kwds)	Log of the cumulative distribution function at k of the given RV.

Continued on next page

Table 5.271 – continued from previous page

sf(k, *args, **kwds)	Survival function (1 - *cdf*) at k of the given RV.
logsf(k, *args, **kwds)	Log of the survival function of the given RV.
ppf(q, *args, **kwds)	Percent point function (inverse of *cdf*) at q of the given RV.
isf(q, *args, **kwds)	Inverse survival function (inverse of *sf*) at q of the given RV.
moment(n, *args, **kwds)	n-th order non-central moment of distribution.
stats(*args, **kwds)	Some statistics of the given RV.
entropy(*args, **kwds)	Differential entropy of the RV.
expect([func, args, loc, lb, ub, ...])	Calculate expected value of a function with respect to the distribution for discrete distribution.
median(*args, **kwds)	Median of the distribution.
mean(*args, **kwds)	Mean of the distribution.
std(*args, **kwds)	Standard deviation of the distribution.
var(*args, **kwds)	Variance of the distribution.
interval(alpha, *args, **kwds)	Confidence interval with equal areas around the median.
__call__(*args, **kwds)	Freeze the distribution for the given arguments.

rv_discrete.**rvs**(**args*, ***kwargs*)

Random variates of given type.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

size : int or tuple of ints, optional

Defining number of random variates (Default is 1). Note that *size* has to be given as keyword, not as positional argument.

random_state : None or int or `np.random.RandomState` instance, optional

If int or RandomState, use it for drawing the random variates. If None, rely on `self.random_state`. Default is None.

Returns **rvs** : ndarray or scalar

Random variates of given *size*.

rv_discrete.**pmf**(*k*, **args*, ***kwds*)

Probability mass function at k of the given RV.

Parameters **k** : array_like

Quantiles.

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

Location parameter (default=0).

Returns **pmf** : array_like

Probability mass function evaluated at k

rv_discrete.**logpmf**(*k*, **args*, ***kwds*)

Log of the probability mass function at k of the given RV.

Parameters **k** : array_like

Quantiles.

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
Location parameter. Default is 0.

Returns **logpmf** : array_like
Log of the probability mass function evaluated at k.

`rv_discrete.cdf`(*k*, **args*, ***kwds*)
Cumulative distribution function of the given RV.

Parameters **k** : array_like, int
Quantiles.

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
Location parameter (default=0).

Returns **cdf** : ndarray
Cumulative distribution function evaluated at *k*.

`rv_discrete.logcdf`(*k*, **args*, ***kwds*)
Log of the cumulative distribution function at k of the given RV.

Parameters **k** : array_like, int
Quantiles.

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
Location parameter (default=0).

Returns **logcdf** : array_like
Log of the cumulative distribution function evaluated at k.

`rv_discrete.sf`(*k*, **args*, ***kwds*)
Survival function (1 - `cdf`) at k of the given RV.

Parameters **k** : array_like
Quantiles.

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
Location parameter (default=0).

Returns **sf** : array_like
Survival function evaluated at k.

`rv_discrete.logsf`(*k*, **args*, ***kwds*)
Log of the survival function of the given RV.

Returns the log of the "survival function," defined as 1 - `cdf`, evaluated at *k*.

Parameters **k** : array_like
Quantiles.

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
Location parameter (default=0).

Returns **logsf** : ndarray
Log of the survival function evaluated at *k*.

`rv_discrete.`**`ppf`**(*q*, **args*, ***kwds*)
Percent point function (inverse of `cdf`) at q of the given RV.

Parameters **q** : array_like
Lower tail probability.
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).

Returns **k** : array_like
Quantile corresponding to the lower tail probability, q.

`rv_discrete.`**`isf`**(*q*, **args*, ***kwds*)
Inverse survival function (inverse of `sf`) at q of the given RV.

Parameters **q** : array_like
Upper tail probability.
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).

Returns **k** : ndarray or scalar
Quantile corresponding to the upper tail probability, q.

`rv_discrete.`**`moment`**(*n*, **args*, ***kwds*)
n-th order non-central moment of distribution.

Parameters **n** : int, n >= 1
Order of moment.
arg1, arg2, arg3,... : float
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

`rv_discrete.`**`stats`**(**args*, ***kwds*)
Some statistics of the given RV.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional (continuous RVs only)
scale parameter (default=1)
moments : str, optional
composed of letters ['mvsk'] defining which moments to compute: 'm' = mean, 'v' = variance, 's' = (Fisher's) skew, 'k' = (Fisher's) kurtosis. (default is 'mv')

Returns **stats** : sequence
of requested moments.

`rv_discrete.`**`entropy`**(**args*, ***kwds*)
Differential entropy of the RV.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional

Location parameter (default=0).

scale : array_like, optional (continuous distributions only).

Scale parameter (default=1).

Notes

Entropy is defined base *e*:

```
>>> drv = rv_discrete(values=((0, 1), (0.5, 0.5)))
>>> np.allclose(drv.entropy(), np.log(2.0))
True
```

`rv_discrete.`**`expect`**(*func=None*, *args=()*, *loc=0*, *lb=None*, *ub=None*, *conditional=False*, *maxcount=1000*, *tolerance=1e-10*, *chunksize=32*)

Calculate expected value of a function with respect to the distribution for discrete distribution.

Parameters

func : callable, optional

Function for which the expectation value is calculated. Takes only one argument. The default is the identity mapping f(k) = k.

args : tuple, optional

Shape parameters of the distribution.

loc : float, optional

Location parameter. Default is 0.

lb, ub : int, optional

Lower and upper bound for the summation, default is set to the support of the distribution, inclusive (`ul <= k <= ub`).

conditional : bool, optional

If true then the expectation is corrected by the conditional probability of the summation interval. The return value is the expectation of the function, *func*, conditional on being in the given interval (k such that `ul <= k <= ub`). Default is False.

maxcount : int, optional

Maximal number of terms to evaluate (to avoid an endless loop for an infinite sum). Default is 1000.

tolerance : float, optional

Absolute tolerance for the summation. Default is 1e-10.

chunksize : int, optional

Iterate over the support of a distributions in chunks of this size. Default is 32.

Returns

expect : float

Expected value.

Notes

For heavy-tailed distributions, the expected value may or may not exist, depending on the function, *func*. If it does exist, but the sum converges slowly, the accuracy of the result may be rather low. For instance, for `zipf(4)`, accuracy for mean, variance in example is only 1e-5. increasing *maxcount* and/or *chunksize* may improve the result, but may also make zipf very slow.

The function is not vectorized.

`rv_discrete.`**`median`**(**args*, ***kwds*)

Median of the distribution.

Parameters

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

Location parameter, Default is 0.

scale : array_like, optional

Scale parameter, Default is 1.

Returns **median** : float

The median of the distribution.

See also:

stats.distributions.rv_discrete.ppf
Inverse of the CDF

rv_discrete.**mean**(**args*, ***kwds*)
Mean of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **mean** : float

the mean of the distribution

rv_discrete.**std**(**args*, ***kwds*)
Standard deviation of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **std** : float

standard deviation of the distribution

rv_discrete.**var**(**args*, ***kwds*)
Variance of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **var** : float

the variance of the distribution

rv_discrete.**interval**(*alpha*, **args*, ***kwds*)
Confidence interval with equal areas around the median.

Parameters **alpha** : array_like of float

Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].

arg1, arg2, ... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
location parameter, Default is 0.

scale : array_like, optional
scale parameter, Default is 1.

Returns **a, b** : ndarray of float
end-points of range that contain `100 * alpha %` of the rv's possible values.

`rv_discrete.__call__(*args, **kwds)`
Freeze the distribution for the given arguments.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

Returns **rv_frozen** : rv_frozen instance
The frozen distribution.

class `scipy.stats.rv_histogram(histogram, *args, **kwargs)`
Generates a distribution given by a histogram. This is useful to generate a template distribution from a binned datasample.

As a subclass of the *rv_continuous* class, *rv_histogram* inherits from it a collection of generic methods (see *rv_continuous* for the full list), and implements them based on the properties of the provided binned datasample.

Parameters **histogram** : tuple of array_like
Tuple containing two array_like objects The first containing the content of n bins The second containing the (n+1) bin boundaries In particular the return value np.histogram is accepted

Notes

There are no additional shape parameters except for the loc and scale. The pdf is defined as a stepwise function from the provided histogram The cdf is a linear interpolation of the pdf.

New in version 0.19.0.

Examples

Create a scipy.stats distribution from a numpy histogram

```
>>> import scipy.stats
>>> import numpy as np
>>> data = scipy.stats.norm.rvs(size=100000, loc=0, scale=1.5, random_state=123)
>>> hist = np.histogram(data, bins=100)
>>> hist_dist = scipy.stats.rv_histogram(hist)
```

Behaves like an ordinary scipy rv_continuous distribution

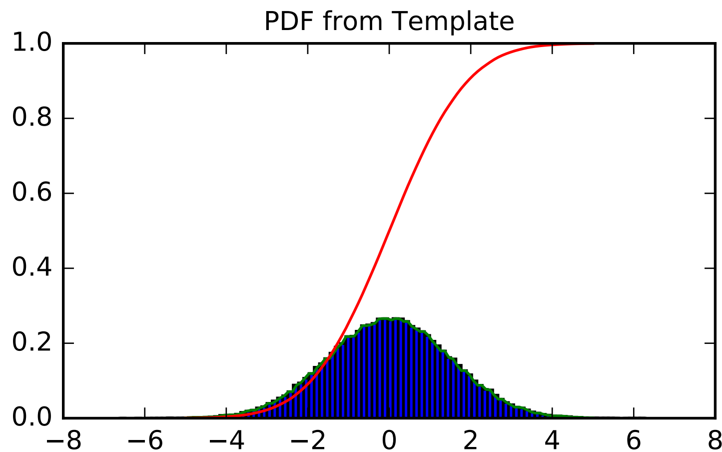
```
>>> hist_dist.pdf(1.0)
0.20538577847618705
>>> hist_dist.cdf(2.0)
0.90818568543056499
```

PDF is zero above (below) the highest (lowest) bin of the histogram, defined by the max (min) of the original dataset

```
>>> hist_dist.pdf(np.max(data))
0.0
>>> hist_dist.cdf(np.max(data))
1.0
>>> hist_dist.pdf(np.min(data))
7.7591907244498314e-05
>>> hist_dist.cdf(np.min(data))
0.0
```

PDF and CDF follow the histogram

```
>>> import matplotlib.pyplot as plt
>>> X = np.linspace(-5.0, 5.0, 100)
>>> plt.title("PDF from Template")
>>> plt.hist(data, normed=True, bins=100)
>>> plt.plot(X, hist_dist.pdf(X), label='PDF')
>>> plt.plot(X, hist_dist.cdf(X), label='CDF')
>>> plt.show()
```



Attributes

random_state	Get or set the RandomState object for generating random variates.

`rv_histogram.`**`random_state`**

Get or set the RandomState object for generating random variates.

This can be either None or an existing RandomState object.

If None (or np.random), use the RandomState singleton used by np.random. If already a RandomState instance, use it. If an int, use a new RandomState instance seeded with seed.

Methods

__call__(*args, **kwds)	Freeze the distribution for the given arguments.

Continued on next page

Table 5.273 – continued from previous page

cdf(x, *args, **kwds)	Cumulative distribution function of the given RV.
entropy(*args, **kwds)	Differential entropy of the RV.
expect([func, args, loc, scale, lb, ub, ...])	Calculate expected value of a function with respect to the distribution.
fit(data, *args, **kwds)	Return MLEs for shape (if applicable), location, and scale parameters from data.
fit_loc_scale(data, *args)	Estimate loc and scale parameters from data using 1st and 2nd moments.
freeze(*args, **kwds)	Freeze the distribution for the given arguments.
interval(alpha, *args, **kwds)	Confidence interval with equal areas around the median.
isf(q, *args, **kwds)	Inverse survival function (inverse of *sf*) at q of the given RV.
logcdf(x, *args, **kwds)	Log of the cumulative distribution function at x of the given RV.
logpdf(x, *args, **kwds)	Log of the probability density function at x of the given RV.
logsf(x, *args, **kwds)	Log of the survival function of the given RV.
mean(*args, **kwds)	Mean of the distribution.
median(*args, **kwds)	Median of the distribution.
moment(n, *args, **kwds)	n-th order non-central moment of distribution.
nnlf(theta, x)	Return negative loglikelihood function.
pdf(x, *args, **kwds)	Probability density function at x of the given RV.
ppf(q, *args, **kwds)	Percent point function (inverse of *cdf*) at q of the given RV.
rvs(*args, **kwds)	Random variates of given type.
sf(x, *args, **kwds)	Survival function (1 - *cdf*) at x of the given RV.
stats(*args, **kwds)	Some statistics of the given RV.
std(*args, **kwds)	Standard deviation of the distribution.
var(*args, **kwds)	Variance of the distribution.

rv_histogram.**__call__**(**args*, ***kwds*)

Freeze the distribution for the given arguments.

Parameters **arg1, arg2, arg3,...** : array_like

The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

Returns **rv_frozen** : rv_frozen instance

The frozen distribution.

rv_histogram.**cdf**(*x*, **args*, ***kwds*)

Cumulative distribution function of the given RV.

Parameters **x** : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **cdf** : ndarray

Cumulative distribution function evaluated at *x*

rv_histogram.**entropy**(**args*, ***kwds*)
Differential entropy of the RV.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).
scale : array_like, optional (continuous distributions only).
Scale parameter (default=1).

Notes

Entropy is defined base *e*:

```
>>> drv = rv_discrete(values=((0, 1), (0.5, 0.5)))
>>> np.allclose(drv.entropy(), np.log(2.0))
True
```

rv_histogram.**expect**(*func=None*, *args=()*, *loc=0*, *scale=1*, *lb=None*, *ub=None*, *conditional=False*, ***kwds*)
Calculate expected value of a function with respect to the distribution.

The expected value of a function `f(x)` with respect to a distribution `dist` is defined as:

```
        ubound
E[x] = Integral(f(x) * dist.pdf(x))
        lbound
```

Parameters **func** : callable, optional
Function for which integral is calculated. Takes only one argument. The default is the identity mapping f(x) = x.
args : tuple, optional
Shape parameters of the distribution.
loc : float, optional
Location parameter (default=0).
scale : float, optional
Scale parameter (default=1).
lb, ub : scalar, optional
Lower and upper bound for integration. Default is set to the support of the distribution.
conditional : bool, optional
If True, the integral is corrected by the conditional probability of the integration interval. The return value is the expectation of the function, conditional on being in the given interval. Default is False.
Additional keyword arguments are passed to the integration routine.

Returns **expect** : float
The calculated expected value.

Notes

The integration behavior of this function is inherited from *integrate.quad*.

rv_histogram.**fit**(*data*, **args*, ***kwds*)
Return MLEs for shape (if applicable), location, and scale parameters from data.

MLE stands for Maximum Likelihood Estimate. Starting estimates for the fit are given by input arguments; for any arguments not provided with starting estimates, `self._fitstart(data)` is called to generate such.

One can hold some parameters fixed to specific values by passing in keyword arguments `f0`, `f1`, ..., `fn` (for shape parameters) and `floc` and `fscale` (for location and scale parameters, respectively).

Parameters

data : array_like
: Data to use in calculating the MLEs.

args : floats, optional
: Starting value(s) for any shape-characterizing arguments (those not provided will be determined by a call to `_fitstart(data)`). No default value.

kwds : floats, optional
: Starting values for the location and scale parameters; no default. Special keyword arguments are recognized as holding certain parameters fixed:
  - f0...fn : hold respective shape parameters fixed. Alternatively, shape parameters to fix can be specified by name. For example, if `self.shapes == "a, b"`, `fa``and ``fix_a` are equivalent to `f0`, and `fb` and `fix_b` are equivalent to `f1`.
  - floc : hold location parameter fixed to specified value.
  - fscale : hold scale parameter fixed to specified value.
  - optimizer : The optimizer to use. The optimizer must take `func`, and starting position as the first two arguments, plus `args` (for extra arguments to pass to the function to be optimized) and `disp=0` to suppress output as keyword arguments.

Returns

mle_tuple : tuple of floats
: MLEs for any shape parameters (if applicable), followed by those for location and scale. For most random variables, shape statistics will be returned, but there are exceptions (e.g. `norm`).

Notes

This fit is computed by maximizing a log-likelihood function, with penalty applied for samples outside of range of the distribution. The returned answer is not guaranteed to be the globally optimal MLE, it may only be locally optimal, or the optimization may fail altogether.

Examples

Generate some data to fit: draw random variates from the *beta* distribution

```
>>> from scipy.stats import beta
>>> a, b = 1., 2.
>>> x = beta.rvs(a, b, size=1000)
```

Now we can fit all four parameters (`a`, `b`, `loc` and `scale`):

```
>>> a1, b1, loc1, scale1 = beta.fit(x)
```

We can also use some prior knowledge about the dataset: let's keep `loc` and `scale` fixed:

```
>>> a1, b1, loc1, scale1 = beta.fit(x, floc=0, fscale=1)
>>> loc1, scale1
(0, 1)
```

We can also keep shape parameters fixed by using `f`-keywords. To keep the zero-th shape parameter `a` equal 1, use `f0=1` or, equivalently, `fa=1`:

```
>>> a1, b1, loc1, scale1 = beta.fit(x, fa=1, floc=0, fscale=1)
>>> a1
1
```

Not all distributions return estimates for the shape parameters. `norm` for example just returns estimates for location and scale:

```
>>> from scipy.stats import norm
>>> x = norm.rvs(a, b, size=1000, random_state=123)
>>> loc1, scale1 = norm.fit(x)
>>> loc1, scale1
(0.92087172783841631, 2.0015750750324668)
```

`rv_histogram.`**`fit_loc_scale`**(*data*, **args*)

Estimate loc and scale parameters from data using 1st and 2nd moments.

Parameters **data** : array_like
Data to fit.

arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

Returns **Lhat** : float
Estimated location parameter for the data.

Shat : float
Estimated scale parameter for the data.

`rv_histogram.`**`freeze`**(**args*, ***kwds*)

Freeze the distribution for the given arguments.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution. Should include all the non-optional arguments, may include `loc` and `scale`.

Returns **rv_frozen** : rv_frozen instance
The frozen distribution.

`rv_histogram.`**`interval`**(*alpha*, **args*, ***kwds*)

Confidence interval with equal areas around the median.

Parameters **alpha** : array_like of float
Probability that an rv will be drawn from the returned range. Each value should be in the range [0, 1].

arg1, arg2, ... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).

loc : array_like, optional
location parameter, Default is 0.

scale : array_like, optional
scale parameter, Default is 1.

Returns **a, b** : ndarray of float
end-points of range that contain `100 * alpha %` of the rv's possible values.

`rv_histogram.`**`isf`**(*q*, **args*, ***kwds*)

Inverse survival function (inverse of `sf`) at q of the given RV.

Parameters **q** : array_like
upper tail probability

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **x** : ndarray or scalar

Quantile corresponding to the upper tail probability q.

`rv_histogram.`**`logcdf`**(*x*, **args*, ***kwds*)

Log of the cumulative distribution function at x of the given RV.

Parameters **x** : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **logcdf** : array_like

Log of the cumulative distribution function evaluated at x

`rv_histogram.`**`logpdf`**(*x*, **args*, ***kwds*)

Log of the probability density function at x of the given RV.

This uses a more numerically accurate calculation if available.

Parameters **x** : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **logpdf** : array_like

Log of the probability density function evaluated at x

`rv_histogram.`**`logsf`**(*x*, **args*, ***kwds*)

Log of the survival function of the given RV.

Returns the log of the "survival function," defined as (1 - `cdf`), evaluated at *x*.

Parameters **x** : array_like

quantiles

arg1, arg2, arg3,... : array_like

The shape parameter(s) for the distribution (see docstring of the instance object for more information)

loc : array_like, optional

location parameter (default=0)

scale : array_like, optional

scale parameter (default=1)

Returns **logsf** : ndarray

Log of the survival function evaluated at *x*.

`rv_histogram.`**`mean`**(**args*, ***kwds*)

Mean of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **mean** : float
the mean of the distribution

`rv_histogram.`**`median`**`(*args, **kwds)`
Median of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
Location parameter, Default is 0.
scale : array_like, optional
Scale parameter, Default is 1.

Returns **median** : float
The median of the distribution.

See also:

`stats.distributions.rv_discrete.ppf`
Inverse of the CDF

`rv_histogram.`**`moment`**`(n, *args, **kwds)`
n-th order non-central moment of distribution.

Parameters **n** : int, n >= 1
Order of moment.
arg1, arg2, arg3,... : float
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

`rv_histogram.`**`nnlf`**`(theta, x)`
Return negative loglikelihood function.

Notes

This is `-sum(log pdf(x, theta), axis=0)` where *theta* are the parameters (including loc and scale).

`rv_histogram.`**`pdf`**`(x, *args, **kwds)`
Probability density function at x of the given RV.

Parameters **x** : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional

Returns **pdf** : ndarray
scale parameter (default=1)
Probability density function evaluated at x

`rv_histogram.`**`ppf`**`(`*q*, **args*, ***kwds*`)`
Percent point function (inverse of `cdf`) at q of the given RV.

Parameters **q** : array_like
lower tail probability
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **x** : array_like
quantile corresponding to the lower tail probability q.

`rv_histogram.`**`rvs`**`(`**args*, ***kwds*`)`
Random variates of given type.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information).
loc : array_like, optional
Location parameter (default=0).
scale : array_like, optional
Scale parameter (default=1).
size : int or tuple of ints, optional
Defining number of random variates (default is 1).
random_state : None or int or `np.random.RandomState` instance, optional
If int or RandomState, use it for drawing the random variates. If None, rely on `self.random_state`. Default is None.

Returns **rvs** : ndarray or scalar
Random variates of given *size*.

`rv_histogram.`**`sf`**`(`*x*, **args*, ***kwds*`)`
Survival function (1 - `cdf`) at x of the given RV.

Parameters **x** : array_like
quantiles
arg1, arg2, arg3,... : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **sf** : array_like
Survival function evaluated at x

`rv_histogram.`**`stats`**`(`**args*, ***kwds*`)`
Some statistics of the given RV.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)

scale : array_like, optional (continuous RVs only)
scale parameter (default=1)
moments : str, optional
composed of letters ['mvsk'] defining which moments to compute: 'm' = mean, 'v' = variance, 's' = (Fisher's) skew, 'k' = (Fisher's) kurtosis. (default is 'mv')

Returns **stats** : sequence
of requested moments.

`rv_histogram.`**`std`**`(*args, **kwds)`
Standard deviation of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **std** : float
standard deviation of the distribution

`rv_histogram.`**`var`**`(*args, **kwds)`
Variance of the distribution.

Parameters **arg1, arg2, arg3,...** : array_like
The shape parameter(s) for the distribution (see docstring of the instance object for more information)
loc : array_like, optional
location parameter (default=0)
scale : array_like, optional
scale parameter (default=1)

Returns **var** : float
the variance of the distribution

5.27.1 Continuous distributions

alpha	An alpha continuous random variable.
anglit	An anglit continuous random variable.
arcsine	An arcsine continuous random variable.
argus	Argus distribution
beta	A beta continuous random variable.
betaprime	A beta prime continuous random variable.
bradford	A Bradford continuous random variable.
burr	A Burr (Type III) continuous random variable.
burr12	A Burr (Type XII) continuous random variable.
cauchy	A Cauchy continuous random variable.
chi	A chi continuous random variable.
chi2	A chi-squared continuous random variable.
cosine	A cosine continuous random variable.
dgamma	A double gamma continuous random variable.
dweibull	A double Weibull continuous random variable.
erlang	An Erlang continuous random variable.
expon	An exponential continuous random variable.

Continued on next page

Table 5.274 – continued from previous page

exponnorm	An exponentially modified Normal continuous random variable.
exponweib	An exponentiated Weibull continuous random variable.
exponpow	An exponential power continuous random variable.
f	An F continuous random variable.
fatiguelife	A fatigue-life (Birnbaum-Saunders) continuous random variable.
fisk	A Fisk continuous random variable.
foldcauchy	A folded Cauchy continuous random variable.
foldnorm	A folded normal continuous random variable.
frechet_r	A Frechet right (or Weibull minimum) continuous random variable.
frechet_l	A Frechet left (or Weibull maximum) continuous random variable.
genlogistic	A generalized logistic continuous random variable.
gennorm	A generalized normal continuous random variable.
genpareto	A generalized Pareto continuous random variable.
genexpon	A generalized exponential continuous random variable.
genextreme	A generalized extreme value continuous random variable.
gausshyper	A Gauss hypergeometric continuous random variable.
gamma	A gamma continuous random variable.
gengamma	A generalized gamma continuous random variable.
genhalflogistic	A generalized half-logistic continuous random variable.
gilbrat	A Gilbrat continuous random variable.
gompertz	A Gompertz (or truncated Gumbel) continuous random variable.
gumbel_r	A right-skewed Gumbel continuous random variable.
gumbel_l	A left-skewed Gumbel continuous random variable.
halfcauchy	A Half-Cauchy continuous random variable.
halflogistic	A half-logistic continuous random variable.
halfnorm	A half-normal continuous random variable.
halfgennorm	The upper half of a generalized normal continuous random variable.
hypsecant	A hyperbolic secant continuous random variable.
invgamma	An inverted gamma continuous random variable.
invgauss	An inverse Gaussian continuous random variable.
invweibull	An inverted Weibull continuous random variable.
johnsonsb	A Johnson SB continuous random variable.
johnsonsu	A Johnson SU continuous random variable.
kappa4	Kappa 4 parameter distribution.
kappa3	Kappa 3 parameter distribution.
ksone	General Kolmogorov-Smirnov one-sided test.
kstwobign	Kolmogorov-Smirnov two-sided test for large N.
laplace	A Laplace continuous random variable.
levy	A Levy continuous random variable.
levy_l	A left-skewed Levy continuous random variable.
levy_stable	A Levy-stable continuous random variable.
logistic	A logistic (or Sech-squared) continuous random variable.
loggamma	A log gamma continuous random variable.
loglaplace	A log-Laplace continuous random variable.

Continued on next page

Table 5.274 – continued from previous page

lognorm	A lognormal continuous random variable.
lomax	A Lomax (Pareto of the second kind) continuous random variable.
maxwell	A Maxwell continuous random variable.
mielke	A Mielke's Beta-Kappa continuous random variable.
nakagami	A Nakagami continuous random variable.
ncx2	A non-central chi-squared continuous random variable.
ncf	A non-central F distribution continuous random variable.
nct	A non-central Student's T continuous random variable.
norm	A normal continuous random variable.
pareto	A Pareto continuous random variable.
pearson3	A pearson type III continuous random variable.
powerlaw	A power-function continuous random variable.
powerlognorm	A power log-normal continuous random variable.
powernorm	A power normal continuous random variable.
rdist	An R-distributed continuous random variable.
reciprocal	A reciprocal continuous random variable.
rayleigh	A Rayleigh continuous random variable.
rice	A Rice continuous random variable.
recipinvgauss	A reciprocal inverse Gaussian continuous random variable.
semicircular	A semicircular continuous random variable.
skewnorm	A skew-normal random variable.
t	A Student's T continuous random variable.
trapz	A trapezoidal continuous random variable.
triang	A triangular continuous random variable.
truncexpon	A truncated exponential continuous random variable.
truncnorm	A truncated normal continuous random variable.
tukeylambda	A Tukey-Lamdba continuous random variable.
uniform	A uniform continuous random variable.
vonmises	A Von Mises continuous random variable.
vonmises_line	A Von Mises continuous random variable.
wald	A Wald continuous random variable.
weibull_min	A Frechet right (or Weibull minimum) continuous random variable.
weibull_max	A Frechet left (or Weibull maximum) continuous random variable.
wrapcauchy	A wrapped Cauchy continuous random variable.

`scipy.stats.`**alpha = <scipy.stats._continuous_distns.alpha_gen object>**

An alpha continuous random variable.

As an instance of the *rv_continuous* class, *alpha* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *alpha* is:

```
alpha.pdf(x, a) = 1/(x**2*Phi(a)*sqrt(2*pi)) * exp(-1/2 * (a-1/x)**2),
```

where `Phi(alpha)` is the normal CDF, `x > 0`, and `a > 0`.

alpha takes `a` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `alpha.pdf(x, a, loc, scale)` is identically equivalent to `alpha.pdf(y, a) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import alpha
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 3.57
>>> mean, var, skew, kurt = alpha.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(alpha.ppf(0.01, a),
...                 alpha.ppf(0.99, a), 100)
>>> ax.plot(x, alpha.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='alpha pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = alpha(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

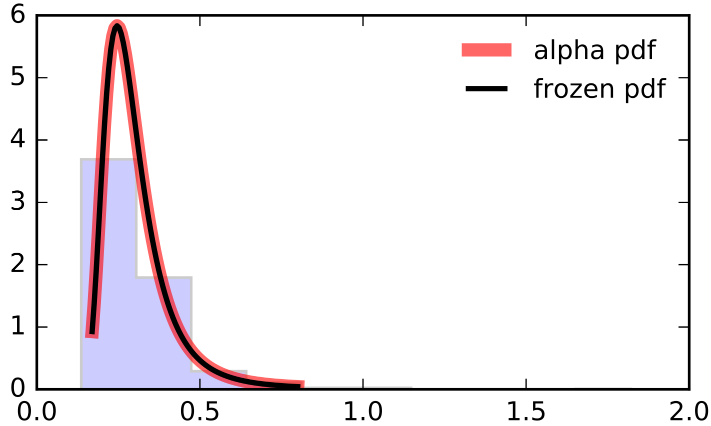
```
>>> vals = alpha.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], alpha.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = alpha.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`anglit`** **= <scipy.stats._continuous_distns.anglit_gen object>**

An anglit continuous random variable.

As an instance of the *`rv_continuous`* class, *`anglit`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `anglit` is:

```
anglit.pdf(x) = sin(2*x + pi/2) = cos(2*x),
```

for `-pi/4 <= x <= pi/4`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `anglit.pdf(x, loc, scale)` is identically equivalent to `anglit.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import anglit
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = anglit.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(anglit.ppf(0.01),
...                 anglit.ppf(0.99), 100)
>>> ax.plot(x, anglit.pdf(x),
...        'r-', lw=5, alpha=0.6, label='anglit pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = anglit()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

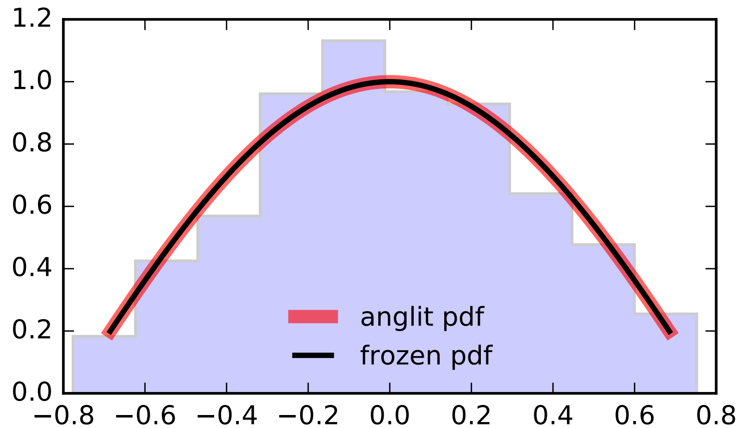
```
>>> vals = anglit.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], anglit.cdf(vals))
True
```

Generate random numbers:

```
>>> r = anglit.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`arcsine`** **= <scipy.stats._continuous_distns.arcsine_gen object>**

An arcsine continuous random variable.

As an instance of the *`rv_continuous`* class, *`arcsine`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `arcsine` is:

```
arcsine.pdf(x) = 1/(pi*sqrt(x*(1-x)))
```

for `0 < x < 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `arcsine.pdf(x, loc, scale)` is identically equivalent to `arcsine.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import arcsine
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = arcsine.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(arcsine.ppf(0.01),
...                 arcsine.ppf(0.99), 100)
>>> ax.plot(x, arcsine.pdf(x),
...        'r-', lw=5, alpha=0.6, label='arcsine pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = arcsine()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

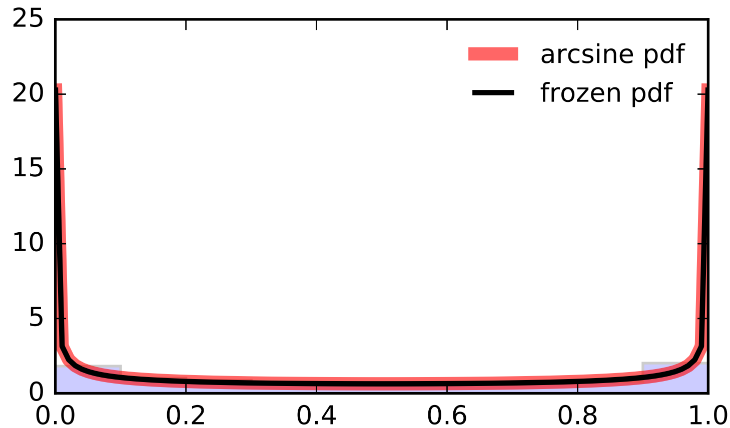
```
>>> vals = arcsine.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], arcsine.cdf(vals))
True
```

Generate random numbers:

```
>>> r = arcsine.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`argus`** **= <scipy.stats._continuous_distns.argus_gen object>**

Argus distribution

As an instance of the *`rv_continuous`* class, *`argus`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `argus` is:

```
argus.pdf(x, chi) = chi**3 / (sqrt(2*pi) * Psi(chi)) * x * sqrt(1-x**2) * exp(- 0.
↪5 * chi**2 * (1 - x**2))

where:
        Psi(chi) = Phi(chi) - chi * phi(chi) - 1/2
        with Phi and phi being the CDF and PDF of a standard normal distribution,
↪ respectively.
```

`argus` takes `chi` as shape a parameter.

References

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `argus.pdf(x, chi, loc, scale)` is identically equivalent to `argus.pdf(y, chi) / scale` with `y = (x - loc) / scale`.

New in version 0.19.0.

[R555]

Examples

```
>>> from scipy.stats import argus
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> chi = 1
>>> mean, var, skew, kurt = argus.stats(chi, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(argus.ppf(0.01, chi),
...                 argus.ppf(0.99, chi), 100)
>>> ax.plot(x, argus.pdf(x, chi),
...        'r-', lw=5, alpha=0.6, label='argus pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = argus(chi)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

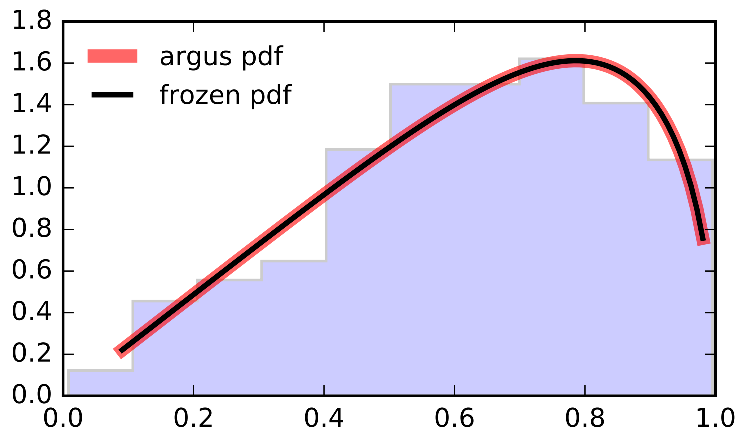
```
>>> vals = argus.ppf([0.001, 0.5, 0.999], chi)
>>> np.allclose([0.001, 0.5, 0.999], argus.cdf(vals, chi))
True
```

Generate random numbers:

```
>>> r = argus.rvs(chi, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(chi, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, chi, loc=0, scale=1)`	Probability density function.
`logpdf(x, chi, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, chi, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, chi, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, chi, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, chi, loc=0, scale=1)`	Log of the survival function.
`ppf(q, chi, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, chi, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, chi, loc=0, scale=1)`	Non-central moment of order n
`stats(chi, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(chi, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, chi, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(chi,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(chi, loc=0, scale=1)`	Median of the distribution.
`mean(chi, loc=0, scale=1)`	Mean of the distribution.
`var(chi, loc=0, scale=1)`	Variance of the distribution.
`std(chi, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, chi, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**beta** = <scipy.stats._continuous_distns.beta_gen object>

A beta continuous random variable.

As an instance of the *rv_continuous* class, *beta* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *beta* is:

```
                     gamma(a+b) * x**(a-1) * (1-x)**(b-1)
beta.pdf(x, a, b) = ------------------------------------
                              gamma(a)*gamma(b)
```

for `0 < x < 1`, `a > 0`, `b > 0`, where `gamma(z)` is the gamma function (*scipy.special.gamma*).

beta takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `beta.pdf(x, a, b, loc, scale)` is identically equivalent to `beta.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import beta
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 2.31, 0.627
>>> mean, var, skew, kurt = beta.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(beta.ppf(0.01, a, b),
...                 beta.ppf(0.99, a, b), 100)
>>> ax.plot(x, beta.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='beta pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a “frozen” RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = beta(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

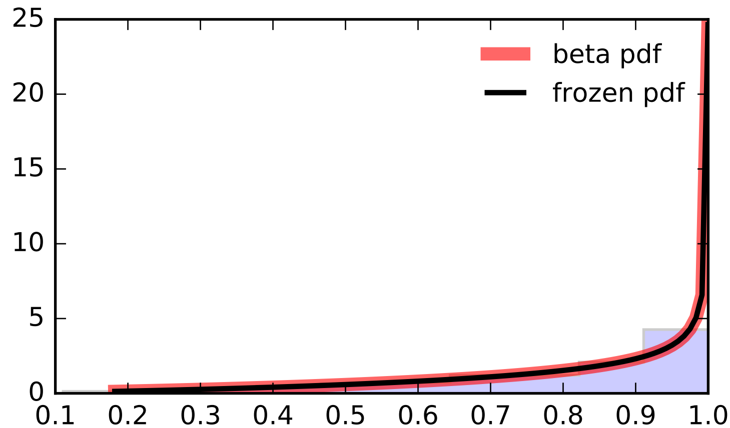
```
>>> vals = beta.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], beta.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = beta.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`betaprime`** **= <scipy.stats._continuous_distns.betaprime_gen object>**

A beta prime continuous random variable.

As an instance of the *`rv_continuous`* class, *`betaprime`* object inherits from it a collection of generic

methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `betaprime` is:

```
betaprime.pdf(x, a, b) = x**(a-1) * (1+x)**(-a-b) / beta(a, b)
```

for `x > 0`, `a > 0`, `b > 0`, where `beta(a, b)` is the beta function (see `scipy.special.beta`).

`betaprime` takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `betaprime.pdf(x, a, b, loc, scale)` is identically equivalent to `betaprime.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import betaprime
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 5, 6
>>> mean, var, skew, kurt = betaprime.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(betaprime.ppf(0.01, a, b),
...                 betaprime.ppf(0.99, a, b), 100)
>>> ax.plot(x, betaprime.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='betaprime pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = betaprime(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

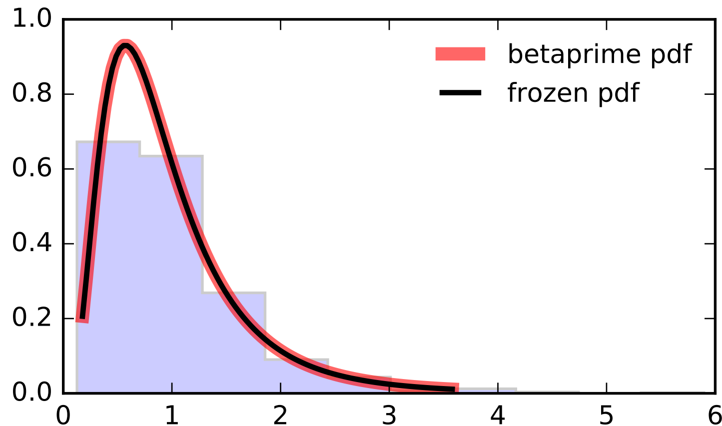
```
>>> vals = betaprime.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], betaprime.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = betaprime.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`bradford`** **= <scipy.stats._continuous_distns.bradford_gen object>**

A Bradford continuous random variable.

As an instance of the *`rv_continuous`* class, *`bradford`* object inherits from it a collection of generic meth-

ods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `bradford` is:

```
bradford.pdf(x, c) = c / (k * (1+c*x)),
```

for `0 < x < 1`, `c > 0` and `k = log(1+c)`.

`bradford` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `bradford.pdf(x, c, loc, scale)` is identically equivalent to `bradford.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import bradford
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.299
>>> mean, var, skew, kurt = bradford.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(bradford.ppf(0.01, c),
...                 bradford.ppf(0.99, c), 100)
>>> ax.plot(x, bradford.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='bradford pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = bradford(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

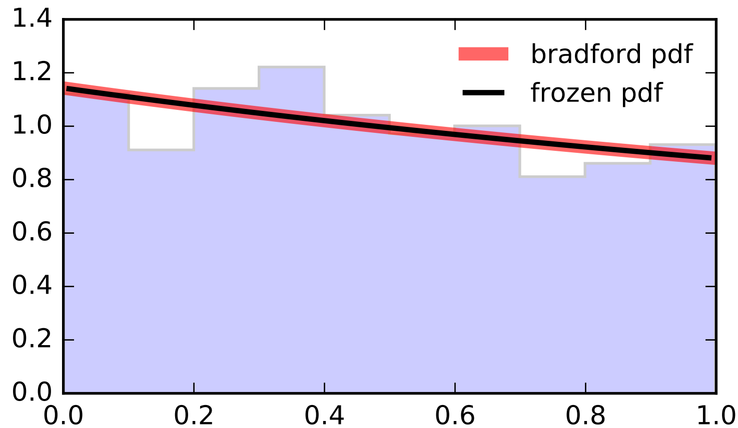
```
>>> vals = bradford.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], bradford.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = bradford.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`burr`** **= <scipy.stats._continuous_distns.burr_gen object>**

A Burr (Type III) continuous random variable.

As an instance of the *rv_continuous* class, *burr* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

fisk a special case of either *burr* or `burr12` with `d = 1`

burr12 Burr Type XII distribution

Notes

The probability density function for *burr* is:

```
burr.pdf(x, c, d) = c * d * x**(-c-1) * (1+x**(-c))**(-d-1)
```

for `x > 0`.

burr takes `c` and `d` as shape parameters.

This is the PDF corresponding to the third CDF given in Burr's list; specifically, it is equation (11) in Burr's paper *[R561]*.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `burr.pdf(x, c, d, loc, scale)` is identically equivalent to `burr.pdf(y, c, d) / scale` with `y = (x - loc) / scale`.

References

[R561]

Examples

```
>>> from scipy.stats import burr
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c, d = 10.5, 4.3
>>> mean, var, skew, kurt = burr.stats(c, d, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(burr.ppf(0.01, c, d),
...                 burr.ppf(0.99, c, d), 100)
>>> ax.plot(x, burr.pdf(x, c, d),
...        'r-', lw=5, alpha=0.6, label='burr pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = burr(c, d)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

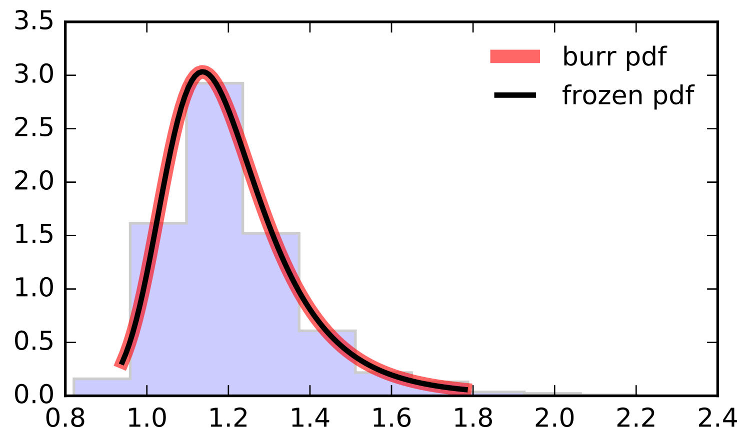
```
>>> vals = burr.ppf([0.001, 0.5, 0.999], c, d)
>>> np.allclose([0.001, 0.5, 0.999], burr.cdf(vals, c, d))
True
```

Generate random numbers:

```
>>> r = burr.rvs(c, d, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, d, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, d, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, d, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, d, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, d, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, d, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, d, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, d, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, d, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, d, loc=0, scale=1)`	Non-central moment of order n
`stats(c, d, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, d, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, d, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c, d), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, d, loc=0, scale=1)`	Median of the distribution.
`mean(c, d, loc=0, scale=1)`	Mean of the distribution.
`var(c, d, loc=0, scale=1)`	Variance of the distribution.
`std(c, d, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, d, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**burr12 = <scipy.stats._continuous_distns.burr12_gen object>**

A Burr (Type XII) continuous random variable.

As an instance of the *rv_continuous* class, *burr12* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

fisk a special case of either *burr* or `burr12` with `d = 1`

burr Burr Type III distribution

Notes

The probability density function for *burr* is:

```
burr12.pdf(x, c, d) = c * d * x**(c-1) * (1+x**(c))**(-d-1)
```

for `x > 0`.

burr12 takes `c` and `d` as shape parameters.

This is the PDF corresponding to the twelfth CDF given in Burr's list; specifically, it is equation (20) in Burr's paper *[R562]*.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `burr12.pdf(x, c, d, loc, scale)` is identically

equivalent to `burr12.pdf(y, c, d) / scale` with `y = (x - loc) / scale`.

The Burr type 12 distribution is also sometimes referred to as the Singh-Maddala distribution from NIST *[R563]*.

References

[R562], *[R563]*

Examples

```
>>> from scipy.stats import burr12
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c, d = 10, 4
>>> mean, var, skew, kurt = burr12.stats(c, d, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(burr12.ppf(0.01, c, d),
...                 burr12.ppf(0.99, c, d), 100)
>>> ax.plot(x, burr12.pdf(x, c, d),
...        'r-', lw=5, alpha=0.6, label='burr12 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = burr12(c, d)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

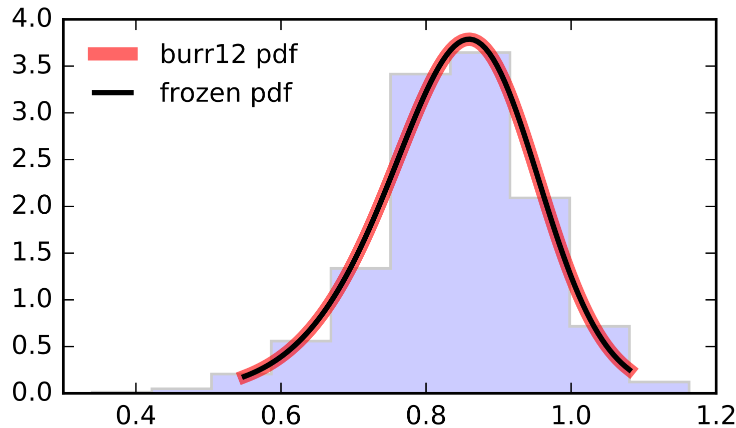
```
>>> vals = burr12.ppf([0.001, 0.5, 0.999], c, d)
>>> np.allclose([0.001, 0.5, 0.999], burr12.cdf(vals, c, d))
True
```

Generate random numbers:

```
>>> r = burr12.rvs(c, d, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, d, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, d, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, d, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, d, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, d, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, d, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, d, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, d, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, d, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, d, loc=0, scale=1)`	Non-central moment of order n
`stats(c, d, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, d, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, d, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c, d), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, d, loc=0, scale=1)`	Median of the distribution.
`mean(c, d, loc=0, scale=1)`	Mean of the distribution.
`var(c, d, loc=0, scale=1)`	Variance of the distribution.
`std(c, d, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, d, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**cauchy = <scipy.stats._continuous_distns.cauchy_gen object>**

A Cauchy continuous random variable.

As an instance of the *rv_continuous* class, *cauchy* object inherits from it a collection of generic methods

(see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `cauchy` is:

```
cauchy.pdf(x) = 1 / (pi * (1 + x**2))
```

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `cauchy.pdf(x, loc, scale)` is identically equivalent to `cauchy.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import cauchy
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = cauchy.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(cauchy.ppf(0.01),
...                 cauchy.ppf(0.99), 100)
>>> ax.plot(x, cauchy.pdf(x),
...        'r-', lw=5, alpha=0.6, label='cauchy pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = cauchy()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

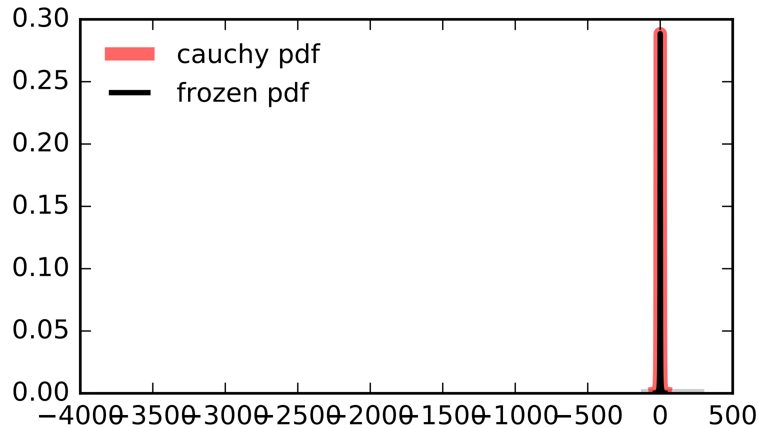
```
>>> vals = cauchy.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], cauchy.cdf(vals))
True
```

Generate random numbers:

```
>>> r = cauchy.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**chi = <scipy.stats._continuous_distns.chi_gen object>**

A chi continuous random variable.

As an instance of the *`rv_continuous`* class, *`chi`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `chi` is:

```
chi.pdf(x, df) = x**(df-1) * exp(-x**2/2) / (2**(df/2-1) * gamma(df/2))
```

for `x > 0`.

Special cases of `chi` are:

- `chi(1, loc, scale)` is equivalent to `halfnorm`
- `chi(2, 0, scale)` is equivalent to `rayleigh`
- `chi(3, 0, scale)` is equivalent to `maxwell`

`chi` takes `df` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `chi.pdf(x, df, loc, scale)` is identically equivalent to `chi.pdf(y, df) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import chi
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> df = 78
>>> mean, var, skew, kurt = chi.stats(df, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(chi.ppf(0.01, df),
...                 chi.ppf(0.99, df), 100)
>>> ax.plot(x, chi.pdf(x, df),
...        'r-', lw=5, alpha=0.6, label='chi pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = chi(df)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

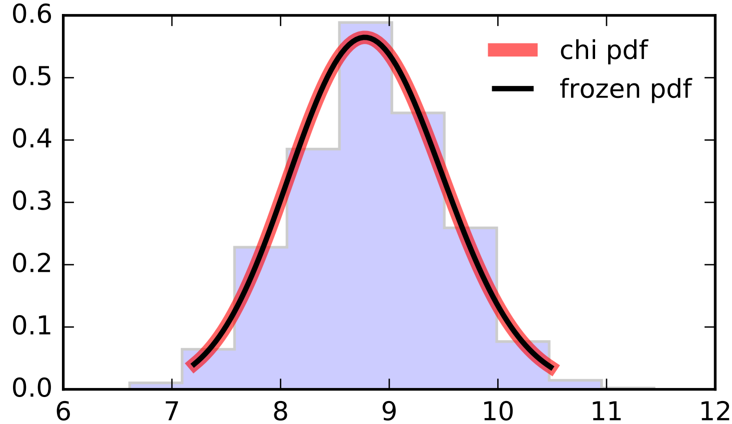
```
>>> vals = chi.ppf([0.001, 0.5, 0.999], df)
>>> np.allclose([0.001, 0.5, 0.999], chi.cdf(vals, df))
True
```

Generate random numbers:

```
>>> r = chi.rvs(df, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(df, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, df, loc=0, scale=1)`	Probability density function.
`logpdf(x, df, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, df, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, df, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, df, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, df, loc=0, scale=1)`	Log of the survival function.
`ppf(q, df, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, df, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, df, loc=0, scale=1)`	Non-central moment of order n
`stats(df, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(df, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, df, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(df,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(df, loc=0, scale=1)`	Median of the distribution.
`mean(df, loc=0, scale=1)`	Mean of the distribution.
`var(df, loc=0, scale=1)`	Variance of the distribution.
`std(df, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, df, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**chi2 = <scipy.stats._continuous_distns.chi2_gen object>**

A chi-squared continuous random variable.

As an instance of the *rv_continuous* class, *chi2* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *chi2* is:

```
chi2.pdf(x, df) = 1 / (2*gamma(df/2)) * (x/2)**(df/2-1) * exp(-x/2)
```

chi2 takes `df` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `chi2.pdf(x, df, loc, scale)` is identically equivalent to `chi2.pdf(y, df) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import chi2
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> df = 55
>>> mean, var, skew, kurt = chi2.stats(df, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(chi2.ppf(0.01, df),
...                 chi2.ppf(0.99, df), 100)
>>> ax.plot(x, chi2.pdf(x, df),
...        'r-', lw=5, alpha=0.6, label='chi2 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = chi2(df)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

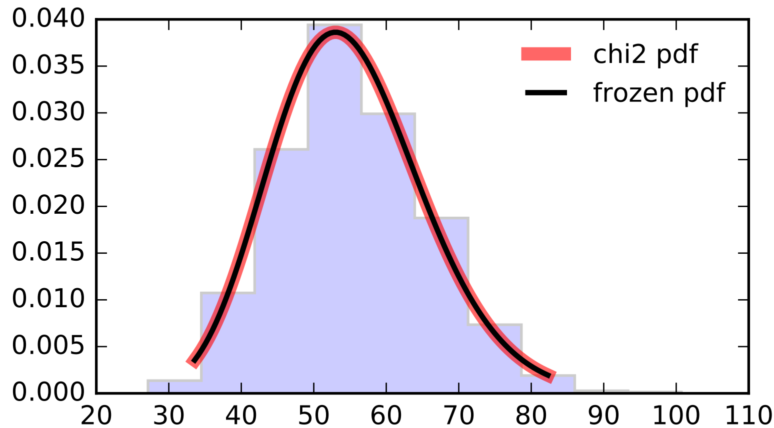
```
>>> vals = chi2.ppf([0.001, 0.5, 0.999], df)
>>> np.allclose([0.001, 0.5, 0.999], chi2.cdf(vals, df))
True
```

Generate random numbers:

```
>>> r = chi2.rvs(df, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(df, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, df, loc=0, scale=1)`	Probability density function.
`logpdf(x, df, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, df, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, df, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, df, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, df, loc=0, scale=1)`	Log of the survival function.
`ppf(q, df, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, df, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, df, loc=0, scale=1)`	Non-central moment of order n
`stats(df, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(df, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, df, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(df,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(df, loc=0, scale=1)`	Median of the distribution.
`mean(df, loc=0, scale=1)`	Mean of the distribution.
`var(df, loc=0, scale=1)`	Variance of the distribution.
`std(df, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, df, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`cosine`** **= <scipy.stats._continuous_distns.cosine_gen object>**

A cosine continuous random variable.

As an instance of the *`rv_continuous`* class, *`cosine`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The cosine distribution is an approximation to the normal distribution. The probability density function for `cosine` is:

```
cosine.pdf(x) = 1/(2*pi) * (1+cos(x))
```

for `-pi <= x <= pi`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `cosine.pdf(x, loc, scale)` is identically equivalent to `cosine.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import cosine
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = cosine.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(cosine.ppf(0.01),
...                 cosine.ppf(0.99), 100)
>>> ax.plot(x, cosine.pdf(x),
...        'r-', lw=5, alpha=0.6, label='cosine pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = cosine()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

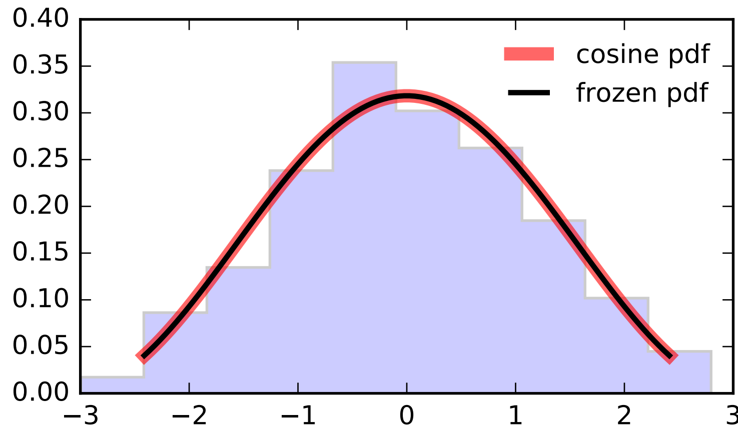
```
>>> vals = cosine.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], cosine.cdf(vals))
True
```

Generate random numbers:

```
>>> r = cosine.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**dgamma = <scipy.stats._continuous_distns.dgamma_gen object>**

A double gamma continuous random variable.

As an instance of the *rv_continuous* class, *dgamma* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *dgamma* is:

```
dgamma.pdf(x, a) = 1 / (2*gamma(a)) * abs(x)**(a-1) * exp(-abs(x))
```

for `a > 0`.

dgamma takes `a` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `dgamma.pdf(x, a, loc, scale)` is identically equivalent to `dgamma.pdf(y, a) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import dgamma
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 1.1
>>> mean, var, skew, kurt = dgamma.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(dgamma.ppf(0.01, a),
...                 dgamma.ppf(0.99, a), 100)
>>> ax.plot(x, dgamma.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='dgamma pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = dgamma(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

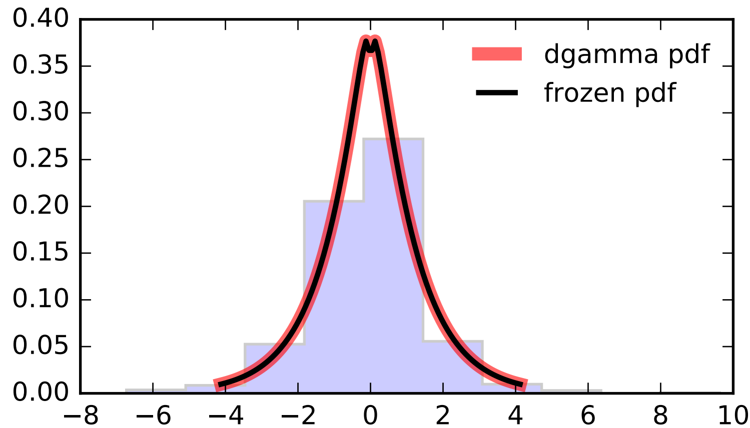
```
>>> vals = dgamma.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], dgamma.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = dgamma.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`dweibull`** **= <scipy.stats._continuous_distns.dweibull_gen object>**

A double Weibull continuous random variable.

As an instance of the *`rv_continuous`* class, *`dweibull`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `dweibull` is:

```
dweibull.pdf(x, c) = c / 2 * abs(x)**(c-1) * exp(-abs(x)**c)
```

`dweibull` takes `d` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `dweibull.pdf(x, c, loc, scale)` is identically equivalent to `dweibull.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import dweibull
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 2.07
>>> mean, var, skew, kurt = dweibull.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(dweibull.ppf(0.01, c),
...                 dweibull.ppf(0.99, c), 100)
>>> ax.plot(x, dweibull.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='dweibull pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = dweibull(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

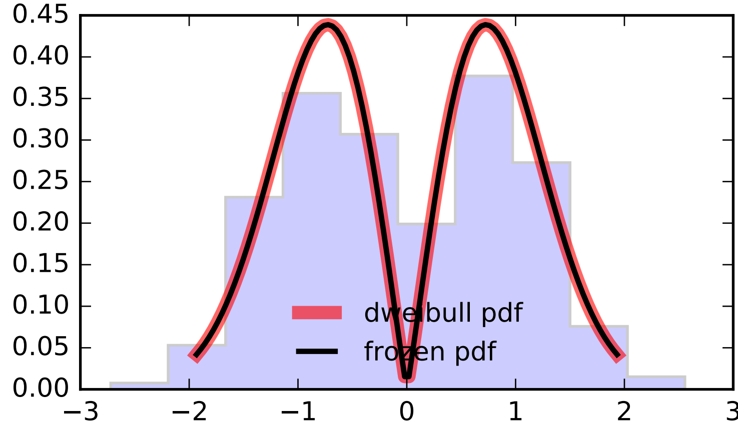
```
>>> vals = dweibull.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], dweibull.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = dweibull.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean(‘m’), variance(‘v’), skew(‘s’), and/or kurtosis(‘k’).
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`erlang`** **= <scipy.stats._continuous_distns.erlang_gen object>**

An Erlang continuous random variable.

As an instance of the *`rv_continuous`* class, *`erlang`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

gamma

Notes

The Erlang distribution is a special case of the Gamma distribution, with the shape parameter *a* an integer. Note that this restriction is not enforced by *erlang*. It will, however, generate a warning the first time a non-integer value is used for the shape parameter.

Refer to *gamma* for examples.

Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**expon = <scipy.stats._continuous_distns.expon_gen object>**

An exponential continuous random variable.

As an instance of the *rv_continuous* class, *expon* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *expon* is:

```
expon.pdf(x) = exp(-x)
```

for `x >= 0`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `expon.pdf(x, loc, scale)` is identically equivalent to

`expon.pdf(y) / scale` with `y = (x - loc) / scale`.

A common parameterization for *expon* is in terms of the rate parameter `lambda`, such that `pdf = lambda * exp(-lambda * x)`. This parameterization corresponds to using `scale = 1 / lambda`.

Examples

```
>>> from scipy.stats import expon
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = expon.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(expon.ppf(0.01),
...                 expon.ppf(0.99), 100)
>>> ax.plot(x, expon.pdf(x),
...        'r-', lw=5, alpha=0.6, label='expon pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = expon()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

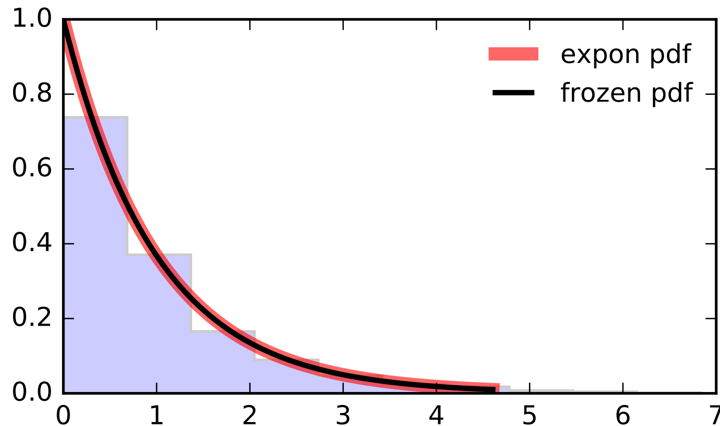
```
>>> vals = expon.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], expon.cdf(vals))
True
```

Generate random numbers:

```
>>> r = expon.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**exponnorm = <scipy.stats._continuous_distns.exponnorm_gen object>**

An exponentially modified Normal continuous random variable.

As an instance of the *rv_continuous* class, *exponnorm* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *exponnorm* is:

```
exponnorm.pdf(x, K) =
    1/(2*K) exp(1/(2 * K**2)) exp(-x / K) * erfc-(x - 1/K) / sqrt(2))
```

where the shape parameter `K > 0`.

It can be thought of as the sum of a normally distributed random value with mean `loc` and sigma `scale` and an exponentially distributed random number with a pdf proportional to `exp(-lambda * x)` where `lambda = (K * scale)**(-1)`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `exponnorm.pdf(x, K, loc, scale)` is identically equivalent to `exponnorm.pdf(y, K) / scale` with `y = (x - loc) / scale`.

An alternative parameterization of this distribution (for example, in Wikipedia) involves three parameters, μ, λ and σ. In the present parameterization this corresponds to having `loc` and `scale` equal to μ and σ, respectively, and shape parameter $K = 1/\sigma\lambda$.

New in version 0.16.0.

Examples

```
>>> from scipy.stats import exponnorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> K = 1.5
>>> mean, var, skew, kurt = exponnorm.stats(K, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(exponnorm.ppf(0.01, K),
...                 exponnorm.ppf(0.99, K), 100)
>>> ax.plot(x, exponnorm.pdf(x, K),
...        'r-', lw=5, alpha=0.6, label='exponnorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = exponnorm(K)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

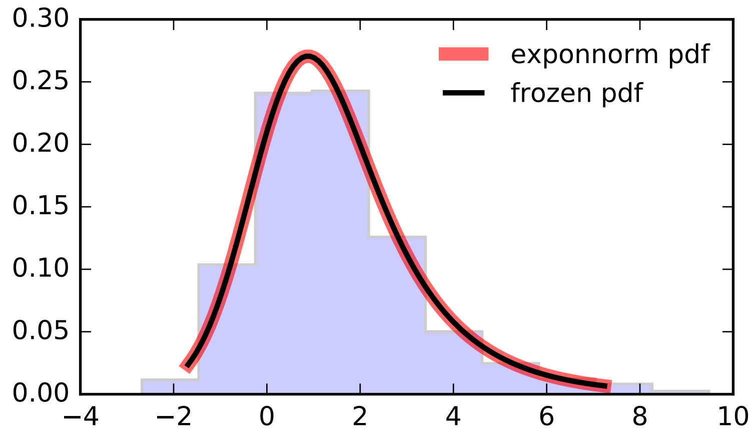
```
>>> vals = exponnorm.ppf([0.001, 0.5, 0.999], K)
>>> np.allclose([0.001, 0.5, 0.999], exponnorm.cdf(vals, K))
True
```

Generate random numbers:

```
>>> r = exponnorm.rvs(K, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(K, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, K, loc=0, scale=1)`	Probability density function.
`logpdf(x, K, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, K, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, K, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, K, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, K, loc=0, scale=1)`	Log of the survival function.
`ppf(q, K, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, K, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, K, loc=0, scale=1)`	Non-central moment of order n
`stats(K, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(K, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, K, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(K,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(K, loc=0, scale=1)`	Median of the distribution.
`mean(K, loc=0, scale=1)`	Mean of the distribution.
`var(K, loc=0, scale=1)`	Variance of the distribution.
`std(K, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, K, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`exponweib`** **= <scipy.stats._continuous_distns.exponweib_gen object>**

An exponentiated Weibull continuous random variable.

As an instance of the *rv_continuous* class, *exponweib* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *exponweib* is:

```
exponweib.pdf(x, a, c) =
    a * c * (1-exp(-x**c))**(a-1) * exp(-x**c)*x**(c-1)
```

for `x > 0`, `a > 0`, `c > 0`.

exponweib takes `a` and `c` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `exponweib.pdf(x, a, c, loc, scale)` is identically equivalent to `exponweib.pdf(y, a, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import exponweib
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, c = 2.89, 1.95
>>> mean, var, skew, kurt = exponweib.stats(a, c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(exponweib.ppf(0.01, a, c),
...                 exponweib.ppf(0.99, a, c), 100)
>>> ax.plot(x, exponweib.pdf(x, a, c),
...        'r-', lw=5, alpha=0.6, label='exponweib pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = exponweib(a, c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

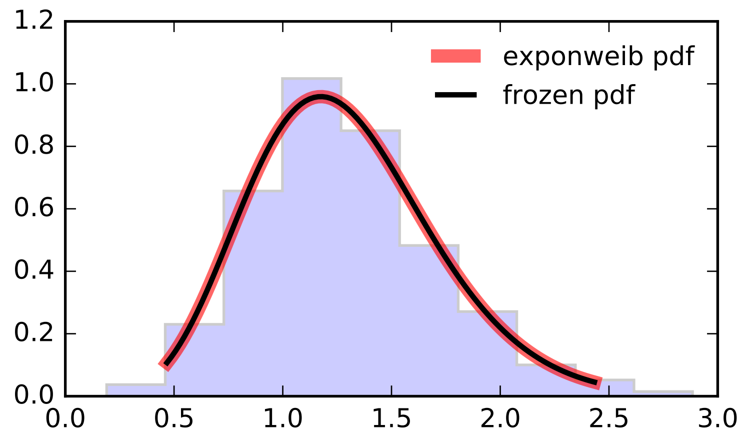
```
>>> vals = exponweib.ppf([0.001, 0.5, 0.999], a, c)
>>> np.allclose([0.001, 0.5, 0.999], exponweib.cdf(vals, a, c))
True
```

Generate random numbers:

```
>>> r = exponweib.rvs(a, c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, c, loc=0, scale=1)`	Non-central moment of order n
`stats(a, c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, c), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, c, loc=0, scale=1)`	Median of the distribution.
`mean(a, c, loc=0, scale=1)`	Mean of the distribution.
`var(a, c, loc=0, scale=1)`	Variance of the distribution.
`std(a, c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**exponpow = <scipy.stats._continuous_distns.exponpow_gen object>**

An exponential power continuous random variable.

As an instance of the *rv_continuous* class, *exponpow* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *exponpow* is:

```
exponpow.pdf(x, b) = b * x**(b-1) * exp(1 + x**b - exp(x**b))
```

for `x >= 0`, `b > 0`. Note that this is a different distribution from the exponential power distribution that is also known under the names "generalized normal" or "generalized Gaussian".

exponpow takes `b` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `exponpow.pdf(x, b, loc, scale)` is identically equivalent to `exponpow.pdf(y, b) / scale` with `y = (x - loc) / scale`.

References

http://www.math.wm.edu/~leemis/chart/UDR/PDFs/Exponentialpower.pdf

Examples

```
>>> from scipy.stats import exponpow
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> b = 2.7
>>> mean, var, skew, kurt = exponpow.stats(b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(exponpow.ppf(0.01, b),
...                 exponpow.ppf(0.99, b), 100)
>>> ax.plot(x, exponpow.pdf(x, b),
...        'r-', lw=5, alpha=0.6, label='exponpow pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = exponpow(b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

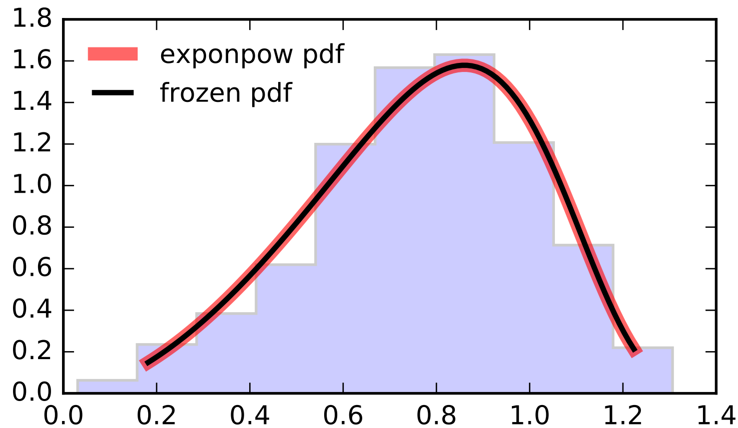
```
>>> vals = exponpow.ppf([0.001, 0.5, 0.999], b)
>>> np.allclose([0.001, 0.5, 0.999], exponpow.cdf(vals, b))
True
```

Generate random numbers:

```
>>> r = exponpow.rvs(b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, b, loc=0, scale=1)`	Non-central moment of order n
`stats(b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(b,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(b, loc=0, scale=1)`	Median of the distribution.
`mean(b, loc=0, scale=1)`	Mean of the distribution.
`var(b, loc=0, scale=1)`	Variance of the distribution.
`std(b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**f = <scipy.stats._continuous_distns.f_gen object>**

An F continuous random variable.

As an instance of the *rv_continuous* class, *f* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `f` is:

```
                     df2**(df2/2) * df1**(df1/2) * x**(df1/2-1)
F.pdf(x, df1, df2) = --------------------------------------------
                     (df2+df1*x)**((df1+df2)/2) * B(df1/2, df2/2)
```

for `x > 0`.

`f` takes `dfn` and `dfd` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `f.pdf(x, dfn, dfd, loc, scale)` is identically equivalent to `f.pdf(y, dfn, dfd) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import f
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> dfn, dfd = 29, 18
>>> mean, var, skew, kurt = f.stats(dfn, dfd, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(f.ppf(0.01, dfn, dfd),
...                 f.ppf(0.99, dfn, dfd), 100)
>>> ax.plot(x, f.pdf(x, dfn, dfd),
...        'r-', lw=5, alpha=0.6, label='f pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = f(dfn, dfd)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

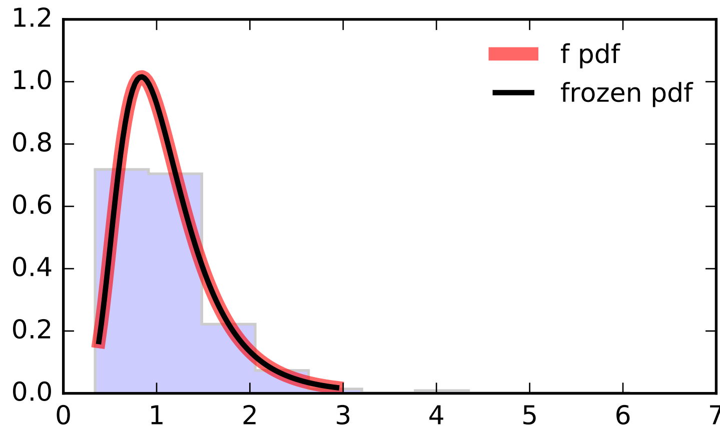
```
>>> vals = f.ppf([0.001, 0.5, 0.999], dfn, dfd)
>>> np.allclose([0.001, 0.5, 0.999], f.cdf(vals, dfn, dfd))
True
```

Generate random numbers:

```
>>> r = f.rvs(dfn, dfd, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(dfn, dfd, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, dfn, dfd, loc=0, scale=1)`	Probability density function.
`logpdf(x, dfn, dfd, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, dfn, dfd, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, dfn, dfd, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, dfn, dfd, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, dfn, dfd, loc=0, scale=1)`	Log of the survival function.
`ppf(q, dfn, dfd, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, dfn, dfd, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, dfn, dfd, loc=0, scale=1)`	Non-central moment of order n
`stats(dfn, dfd, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(dfn, dfd, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, dfn, dfd, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(dfn, dfd), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(dfn, dfd, loc=0, scale=1)`	Median of the distribution.
`mean(dfn, dfd, loc=0, scale=1)`	Mean of the distribution.
`var(dfn, dfd, loc=0, scale=1)`	Variance of the distribution.
`std(dfn, dfd, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, dfn, dfd, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`fatiguelife`** **= <scipy.stats._continuous_distns.fatiguelife_gen object>**

A fatigue-life (Birnbaum-Saunders) continuous random variable.

As an instance of the *`rv_continuous`* class, *`fatiguelife`* object inherits from it a collection of generic

methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *fatiguelife* is:

```
fatiguelife.pdf(x, c) =
    (x+1) / (2*c*sqrt(2*pi*x**3)) * exp(-(x-1)**2/(2*x*c**2))
```

for `x > 0`.

fatiguelife takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `fatiguelife.pdf(x, c, loc, scale)` is identically equivalent to `fatiguelife.pdf(y, c) / scale` with `y = (x - loc) / scale`.

References

[R577]

Examples

```
>>> from scipy.stats import fatiguelife
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 29
>>> mean, var, skew, kurt = fatiguelife.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(fatiguelife.ppf(0.01, c),
...                 fatiguelife.ppf(0.99, c), 100)
>>> ax.plot(x, fatiguelife.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='fatiguelife pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = fatiguelife(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

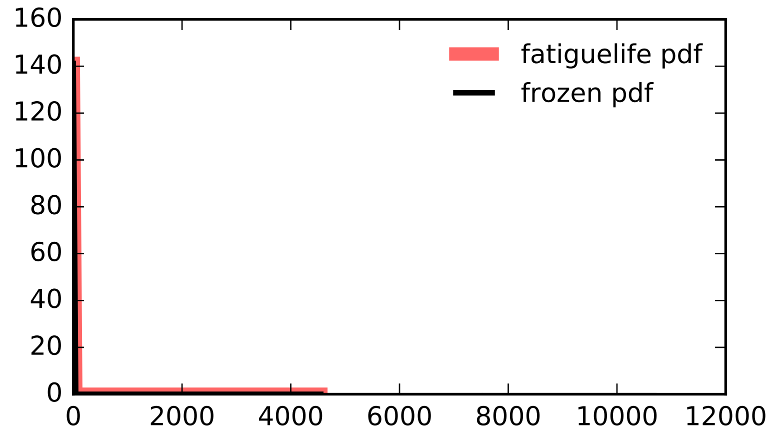
```
>>> vals = fatiguelife.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], fatiguelife.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = fatiguelife.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`fisk`** **= <scipy.stats._continuous_distns.fisk_gen object>**

A Fisk continuous random variable.

The Fisk distribution is also known as the log-logistic distribution, and equals the Burr distribution with `d == 1`.

fisk takes `c` as a shape parameter.

As an instance of the *rv_continuous* class, *fisk* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

burr

Notes

The probability density function for *fisk* is:

```
fisk.pdf(x, c) = c * x**(-c-1) * (1 + x**(-c))**(-2)
```

for `x > 0`.

fisk takes `c` as a shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `fisk.pdf(x, c, loc, scale)` is identically equivalent to `fisk.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import fisk
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 3.09
>>> mean, var, skew, kurt = fisk.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(fisk.ppf(0.01, c),
...                 fisk.ppf(0.99, c), 100)
>>> ax.plot(x, fisk.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='fisk pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = fisk(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

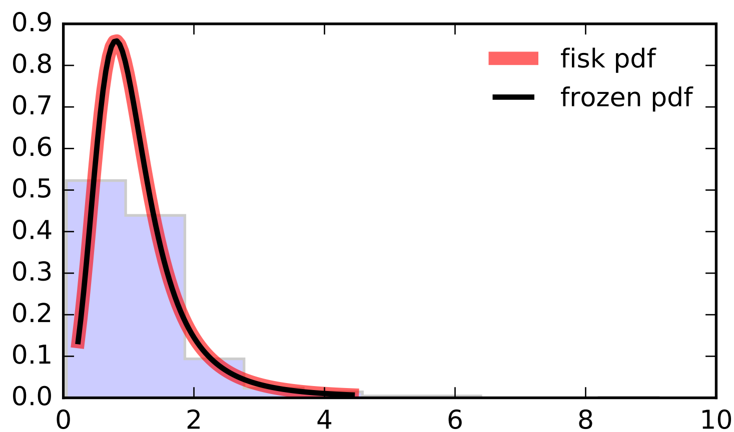
```
>>> vals = fisk.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], fisk.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = fisk.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**foldcauchy = <scipy.stats._continuous_distns.foldcauchy_gen object>**

A folded Cauchy continuous random variable.

As an instance of the *rv_continuous* class, *foldcauchy* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *foldcauchy* is:

```
foldcauchy.pdf(x, c) = 1/(pi*(1+(x-c)**2)) + 1/(pi*(1+(x+c)**2))
```

for `x >= 0`.

foldcauchy takes `c` as a shape parameter.

Examples

```
>>> from scipy.stats import foldcauchy
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 4.72
>>> mean, var, skew, kurt = foldcauchy.stats(c, moments='mvsk')
```

Display the probability density function (pdf):

```
>>> x = np.linspace(foldcauchy.ppf(0.01, c),
...                 foldcauchy.ppf(0.99, c), 100)
>>> ax.plot(x, foldcauchy.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='foldcauchy pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen pdf:

```
>>> rv = foldcauchy(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of cdf and ppf:

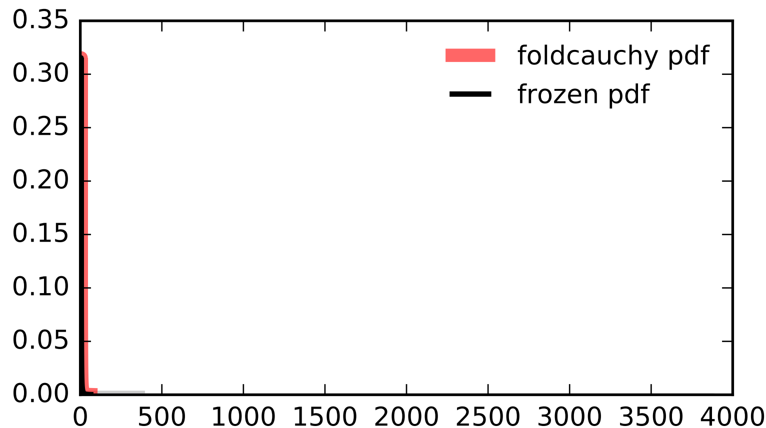
```
>>> vals = foldcauchy.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], foldcauchy.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = foldcauchy.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`foldnorm`** **= <scipy.stats._continuous_distns.foldnorm_gen object>**

A folded normal continuous random variable.

As an instance of the *`rv_continuous`* class, *`foldnorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *`foldnorm`* is:

```
foldnormal.pdf(x, c) = sqrt(2/pi) * cosh(c*x) * exp(-(x**2+c**2)/2)
```

for `c >= 0`.

`foldnorm` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `foldnorm.pdf(x, c, loc, scale)` is identically equivalent to `foldnorm.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import foldnorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 1.95
>>> mean, var, skew, kurt = foldnorm.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(foldnorm.ppf(0.01, c),
...                 foldnorm.ppf(0.99, c), 100)
>>> ax.plot(x, foldnorm.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='foldnorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = foldnorm(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

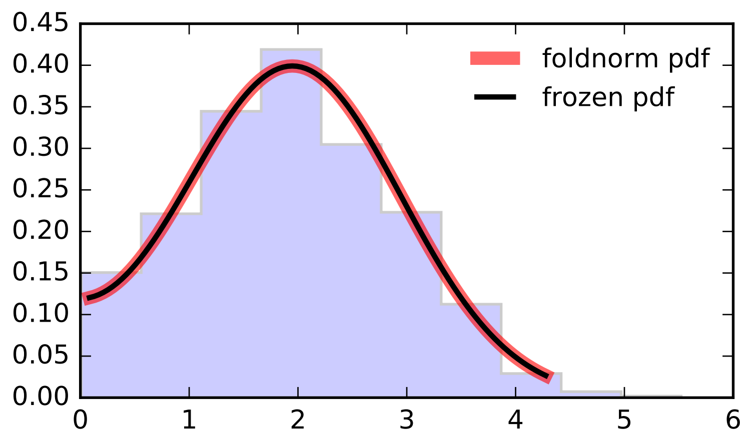
```
>>> vals = foldnorm.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], foldnorm.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = foldnorm.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**frechet_r = <scipy.stats._continuous_distns.frechet_r_gen object>**

A Frechet right (or Weibull minimum) continuous random variable.

As an instance of the *rv_continuous* class, *frechet_r* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

weibull_min
The same distribution as *frechet_r*.

frechet_l, *weibull_max*

Notes

The probability density function for *frechet_r* is:

```
frechet_r.pdf(x, c) = c * x**(c-1) * exp(-x**c)
```

for `x > 0`, `c > 0`.

frechet_r takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `frechet_r.pdf(x, c, loc, scale)` is identically equivalent to `frechet_r.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import frechet_r
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 1.89
>>> mean, var, skew, kurt = frechet_r.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(frechet_r.ppf(0.01, c),
...                 frechet_r.ppf(0.99, c), 100)
>>> ax.plot(x, frechet_r.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='frechet_r pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = frechet_r(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

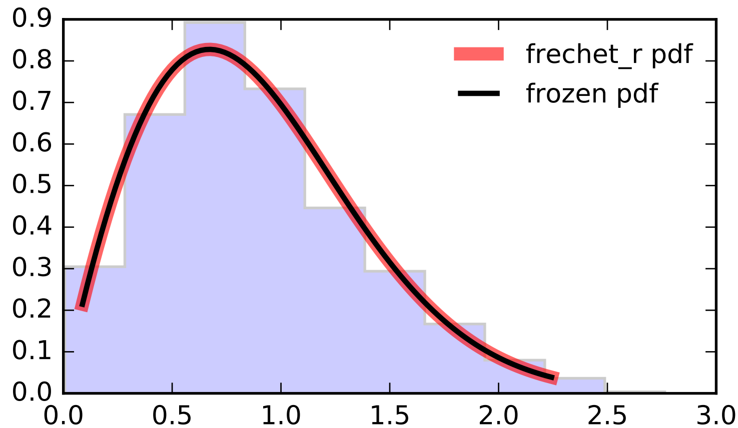
```
>>> vals = frechet_r.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], frechet_r.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = frechet_r.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`frechet_l`** **= <scipy.stats._continuous_distns.frechet_l_gen object>**

A Frechet left (or Weibull maximum) continuous random variable.

As an instance of the *`rv_continuous`* class, *`frechet_l`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

weibull_max
The same distribution as *frechet_l*.

frechet_r, *weibull_min*

Notes

The probability density function for *frechet_l* is:

```
frechet_l.pdf(x, c) = c * (-x)**(c-1) * exp(-(-x)**c)
```

for `x < 0`, `c > 0`.

frechet_l takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `frechet_l.pdf(x, c, loc, scale)` is identically equivalent to `frechet_l.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import frechet_l
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 3.63
>>> mean, var, skew, kurt = frechet_l.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(frechet_l.ppf(0.01, c),
...                 frechet_l.ppf(0.99, c), 100)
>>> ax.plot(x, frechet_l.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='frechet_l pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = frechet_l(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

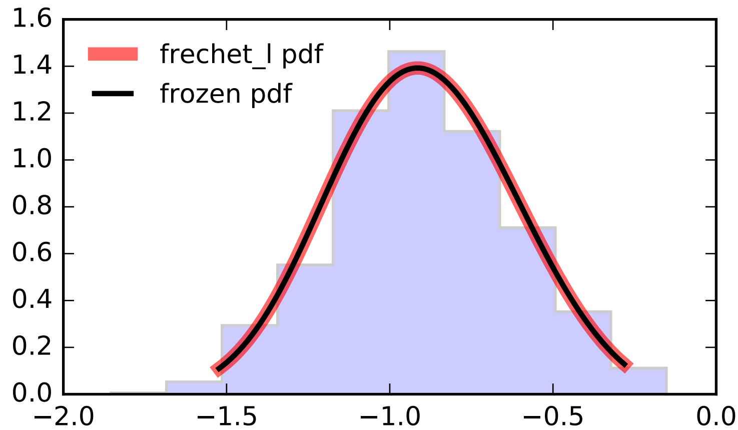
```
>>> vals = frechet_l.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], frechet_l.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = frechet_l.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`genlogistic`** **= <scipy.stats._continuous_distns.genlogistic_gen object>**

A generalized logistic continuous random variable.

As an instance of the *rv_continuous* class, *genlogistic* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *genlogistic* is:

```
genlogistic.pdf(x, c) = c * exp(-x) / (1 + exp(-x))**(c+1)
```

for `x > 0`, `c > 0`.

genlogistic takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `genlogistic.pdf(x, c, loc, scale)` is identically equivalent to `genlogistic.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import genlogistic
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.412
>>> mean, var, skew, kurt = genlogistic.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(genlogistic.ppf(0.01, c),
...                 genlogistic.ppf(0.99, c), 100)
>>> ax.plot(x, genlogistic.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='genlogistic pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = genlogistic(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

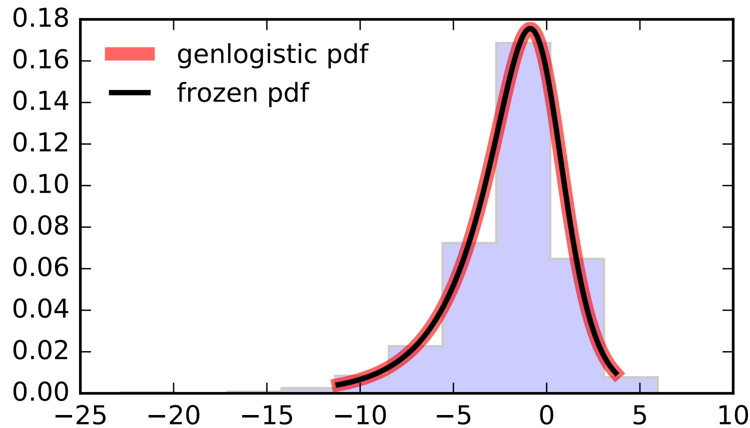
```
>>> vals = genlogistic.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], genlogistic.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = genlogistic.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

rvs(c, loc=0, scale=1, size=1, random_state=None)	Random variates.
pdf(x, c, loc=0, scale=1)	Probability density function.
logpdf(x, c, loc=0, scale=1)	Log of the probability density function.
cdf(x, c, loc=0, scale=1)	Cumulative distribution function.
logcdf(x, c, loc=0, scale=1)	Log of the cumulative distribution function.
sf(x, c, loc=0, scale=1)	Survival function (also defined as 1 - cdf, but *sf* is sometimes more accurate).
logsf(x, c, loc=0, scale=1)	Log of the survival function.
ppf(q, c, loc=0, scale=1)	Percent point function (inverse of cdf — percentiles).
isf(q, c, loc=0, scale=1)	Inverse survival function (inverse of sf).
moment(n, c, loc=0, scale=1)	Non-central moment of order n
stats(c, loc=0, scale=1, moments='mv')	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
entropy(c, loc=0, scale=1)	(Differential) entropy of the RV.
fit(data, c, loc=0, scale=1)	Parameter estimates for generic data.
expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)	Expected value of a function (of one argument) with respect to the distribution.
median(c, loc=0, scale=1)	Median of the distribution.
mean(c, loc=0, scale=1)	Mean of the distribution.
var(c, loc=0, scale=1)	Variance of the distribution.
std(c, loc=0, scale=1)	Standard deviation of the distribution.
interval(alpha, c, loc=0, scale=1)	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**gennorm = <scipy.stats._continuous_distns.gennorm_gen object>**

A generalized normal continuous random variable.

As an instance of the *rv_continuous* class, *gennorm* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

laplace Laplace distribution
norm normal distribution

Notes

The probability density function for *gennorm* is *[R587]*:

```
                                beta
gennorm.pdf(x, beta) =  ---------------   exp(-|x|**beta)
                        2 gamma(1/beta)
```

gennorm takes `beta` as a shape parameter. For `beta = 1`, it is identical to a Laplace distribution. For `beta = 2`, it is identical to a normal distribution (with `scale=1/sqrt(2)`).

References

[R587]

Examples

```
>>> from scipy.stats import gennorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> beta = 1.3
>>> mean, var, skew, kurt = gennorm.stats(beta, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gennorm.ppf(0.01, beta),
...                 gennorm.ppf(0.99, beta), 100)
>>> ax.plot(x, gennorm.pdf(x, beta),
...        'r-', lw=5, alpha=0.6, label='gennorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gennorm(beta)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

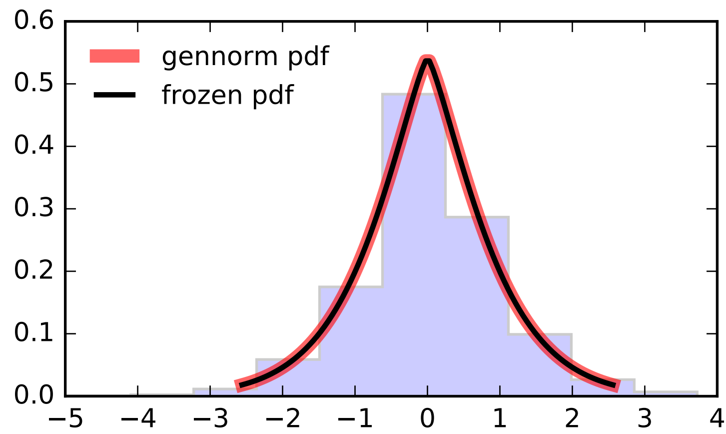
```
>>> vals = gennorm.ppf([0.001, 0.5, 0.999], beta)
>>> np.allclose([0.001, 0.5, 0.999], gennorm.cdf(vals, beta))
True
```

Generate random numbers:

```
>>> r = gennorm.rvs(beta, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(beta, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, beta, loc=0, scale=1)`	Probability density function.
`logpdf(x, beta, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, beta, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, beta, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, beta, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, beta, loc=0, scale=1)`	Log of the survival function.
`ppf(q, beta, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, beta, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, beta, loc=0, scale=1)`	Non-central moment of order n
`stats(beta, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(beta, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, beta, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(beta,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(beta, loc=0, scale=1)`	Median of the distribution.
`mean(beta, loc=0, scale=1)`	Mean of the distribution.
`var(beta, loc=0, scale=1)`	Variance of the distribution.
`std(beta, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, beta, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`genpareto`** **= <scipy.stats._continuous_distns.genpareto_gen object>**

A generalized Pareto continuous random variable.

As an instance of the *`rv_continuous`* class, *`genpareto`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *`genpareto`* is:

```
genpareto.pdf(x, c) = (1 + c * x)**(-1 - 1/c)
```

defined for `x >= 0` if `c >=0`, and for `0 <= x <= -1/c` if `c < 0`.

`genpareto` takes `c` as a shape parameter.

For `c == 0`, *`genpareto`* reduces to the exponential distribution, *`expon`*:

```
genpareto.pdf(x, c=0) = exp(-x)
```

For `c == -1`, *`genpareto`* is uniform on `[0, 1]`:

```
genpareto.cdf(x, c=-1) = x
```

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `genpareto.pdf(x, c, loc, scale)` is identically equivalent to `genpareto.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import genpareto
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.1
>>> mean, var, skew, kurt = genpareto.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(genpareto.ppf(0.01, c),
...                 genpareto.ppf(0.99, c), 100)
>>> ax.plot(x, genpareto.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='genpareto pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = genpareto(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

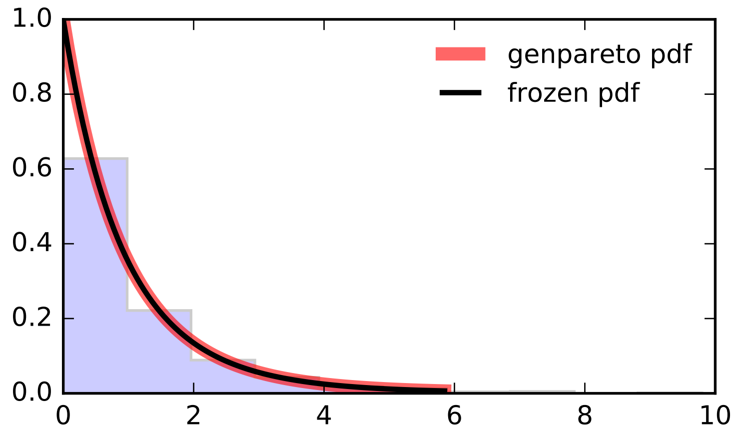
```
>>> vals = genpareto.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], genpareto.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = genpareto.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**genexpon = <scipy.stats._continuous_distns.genexpon_gen object>**

A generalized exponential continuous random variable.

As an instance of the *rv_continuous* class, *genexpon* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *genexpon* is:

```
genexpon.pdf(x, a, b, c) = (a + b * (1 - exp(-c*x))) * 
↪        exp(-a*x - b*x + b/c * (1-exp(-c*x)))
```

for `x >= 0`, `a, b, c > 0`.

genexpon takes `a`, `b` and `c` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `genexpon.pdf(x, a, b, c, loc, scale)` is identically equivalent to `genexpon.pdf(y, a, b, c) / scale` with `y = (x - loc) / scale`.

References

H.K. Ryu, "An Extension of Marshall and Olkin's Bivariate Exponential Distribution", Journal of the American Statistical Association, 1993.

N. Balakrishnan, "The Exponential Distribution: Theory, Methods and Applications", Asit P. Basu.

Examples

```
>>> from scipy.stats import genexpon
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b, c = 9.13, 16.2, 3.28
>>> mean, var, skew, kurt = genexpon.stats(a, b, c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(genexpon.ppf(0.01, a, b, c),
...                 genexpon.ppf(0.99, a, b, c), 100)
>>> ax.plot(x, genexpon.pdf(x, a, b, c),
...        'r-', lw=5, alpha=0.6, label='genexpon pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = genexpon(a, b, c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

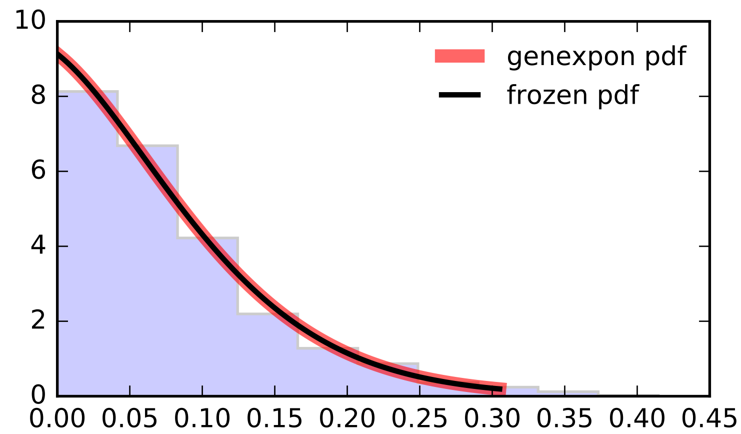
```
>>> vals = genexpon.ppf([0.001, 0.5, 0.999], a, b, c)
>>> np.allclose([0.001, 0.5, 0.999], genexpon.cdf(vals, a, b, c))
True
```

Generate random numbers:

```
>>> r = genexpon.rvs(a, b, c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, c, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b, c), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, c, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, c, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, c, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**genextreme = <scipy.stats._continuous_distns.genextreme_gen object>**

A generalized extreme value continuous random variable.

As an instance of the *rv_continuous* class, *genextreme* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

gumbel_r

Notes

For `c=0`, *genextreme* is equal to *gumbel_r*. The probability density function for *genextreme* is:

```
genextreme.pdf(x, c) =
    exp(-exp(-x))*exp(-x),                    for c==0
    exp(-(1-c*x)**(1/c))*(1-c*x)**(1/c-1),    for x <= 1/c, c > 0
```

Note that several sources and software packages use the opposite convention for the sign of the shape parameter `c`.

genextreme takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `genextreme.pdf(x, c, loc, scale)` is identically equivalent to `genextreme.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import genextreme
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = -0.1
>>> mean, var, skew, kurt = genextreme.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(genextreme.ppf(0.01, c),
...                 genextreme.ppf(0.99, c), 100)
>>> ax.plot(x, genextreme.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='genextreme pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = genextreme(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

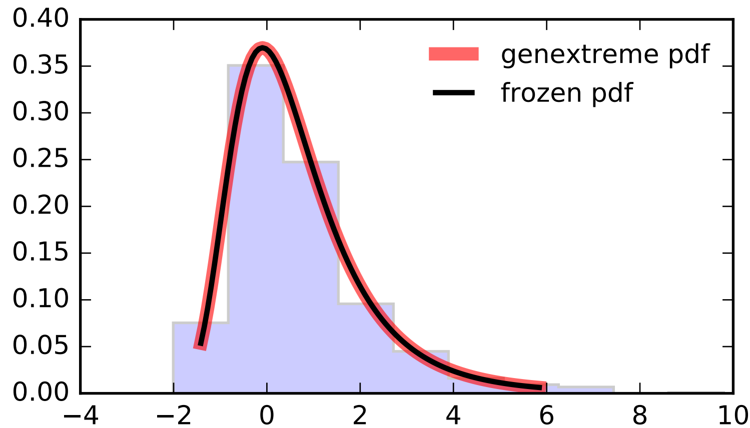
```
>>> vals = genextreme.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], genextreme.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = genextreme.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`gausshyper`** **= <scipy.stats._continuous_distns.gausshyper_gen object>**

A Gauss hypergeometric continuous random variable.

As an instance of the *`rv_continuous`* class, *`gausshyper`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `gausshyper` is:

```
gausshyper.pdf(x, a, b, c, z) =
    C * x**(a-1) * (1-x)**(b-1) * (1+z*x)**(-c)
```

for `0 <= x <= 1`, `a > 0`, `b > 0`, and `C = 1 / (B(a, b) F[2, 1](c, a; a+b; -z))`

`gausshyper` takes `a`, `b`, `c` and `z` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gausshyper.pdf(x, a, b, c, z, loc, scale)` is identically equivalent to `gausshyper.pdf(y, a, b, c, z) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gausshyper
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b, c, z = 13.8, 3.12, 2.51, 5.18
>>> mean, var, skew, kurt = gausshyper.stats(a, b, c, z, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gausshyper.ppf(0.01, a, b, c, z),
...                 gausshyper.ppf(0.99, a, b, c, z), 100)
>>> ax.plot(x, gausshyper.pdf(x, a, b, c, z),
...        'r-', lw=5, alpha=0.6, label='gausshyper pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gausshyper(a, b, c, z)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

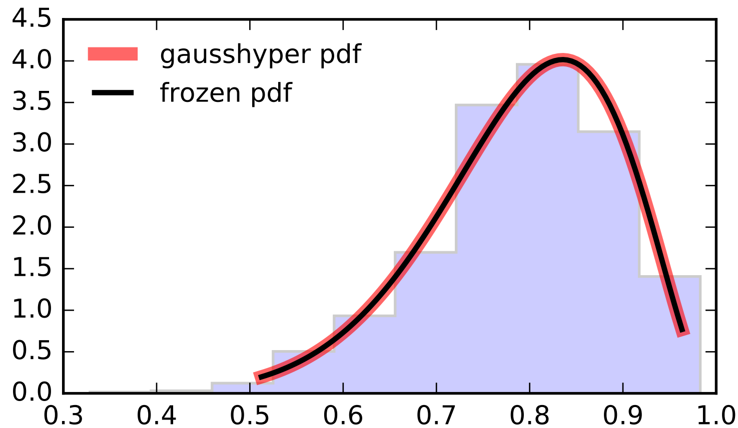
```
>>> vals = gausshyper.ppf([0.001, 0.5, 0.999], a, b, c, z)
>>> np.allclose([0.001, 0.5, 0.999], gausshyper.cdf(vals, a, b, c, z))
True
```

Generate random numbers:

```
>>> r = gausshyper.rvs(a, b, c, z, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, c, z, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, c, z, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, c, z, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, c, z, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, c, z, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, c, z, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, c, z, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, c, z, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, c, z, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, c, z, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, c, z, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, c, z, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, c, z, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b, c, z), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, c, z, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, c, z, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, c, z, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, c, z, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, c, z, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**gamma = <scipy.stats._continuous_distns.gamma_gen object>**

A gamma continuous random variable.

As an instance of the `rv_continuous` class, `gamma` object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

`erlang`, `expon`

Notes

The probability density function for `gamma` is:

```
gamma.pdf(x, a) = x**(a-1) * exp(-x) / gamma(a)
```

for `x >= 0`, `a > 0`. Here `gamma(a)` refers to the gamma function.

`gamma` has a shape parameter *a* which needs to be set explicitly.

When `a` is an integer, `gamma` reduces to the Erlang distribution, and when `a=1` to the exponential distribution.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gamma.pdf(x, a, loc, scale)` is identically equivalent to `gamma.pdf(y, a) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gamma
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 1.99
>>> mean, var, skew, kurt = gamma.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gamma.ppf(0.01, a),
...                 gamma.ppf(0.99, a), 100)
>>> ax.plot(x, gamma.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='gamma pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gamma(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

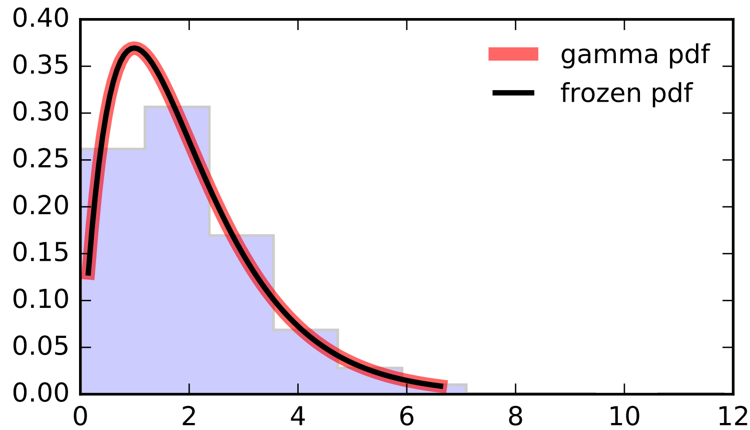
```
>>> vals = gamma.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], gamma.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = gamma.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**gengamma = <scipy.stats._continuous_distns.gengamma_gen object>**

A generalized gamma continuous random variable.

As an instance of the *rv_continuous* class, *gengamma* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *gengamma* is:

```
gengamma.pdf(x, a, c) = abs(c) * x**(c*a-1) * exp(-x**c) / gamma(a)
```

for `x >= 0`, `a > 0`, and `c != 0`.

gengamma takes `a` and `c` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gengamma.pdf(x, a, c, loc, scale)` is identically equivalent to `gengamma.pdf(y, a, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gengamma
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, c = 4.42, -3.12
>>> mean, var, skew, kurt = gengamma.stats(a, c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gengamma.ppf(0.01, a, c),
...                 gengamma.ppf(0.99, a, c), 100)
>>> ax.plot(x, gengamma.pdf(x, a, c),
...        'r-', lw=5, alpha=0.6, label='gengamma pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gengamma(a, c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

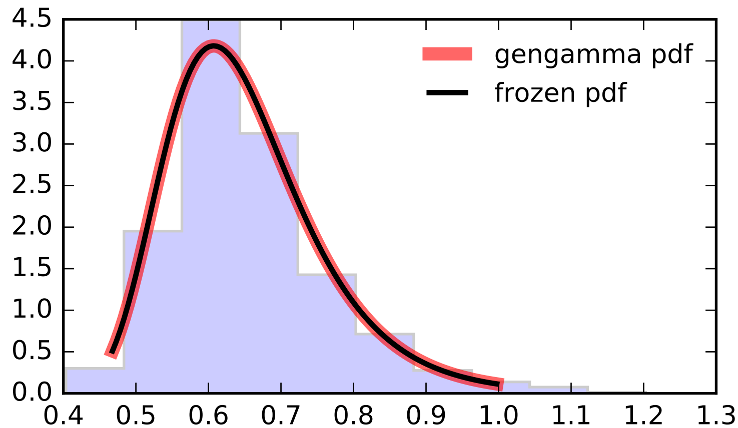
```
>>> vals = gengamma.ppf([0.001, 0.5, 0.999], a, c)
>>> np.allclose([0.001, 0.5, 0.999], gengamma.cdf(vals, a, c))
True
```

Generate random numbers:

```
>>> r = gengamma.rvs(a, c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, c, loc=0, scale=1)`	Non-central moment of order n
`stats(a, c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, c), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, c, loc=0, scale=1)`	Median of the distribution.
`mean(a, c, loc=0, scale=1)`	Mean of the distribution.
`var(a, c, loc=0, scale=1)`	Variance of the distribution.
`std(a, c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`genhalflogistic`** **= <scipy.stats._continuous_distns.genhalflogistic_gen object>**

A generalized half-logistic continuous random variable.

As an instance of the *`rv_continuous`* class, *`genhalflogistic`* object inherits from it a collection of

generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *genhalflogistic* is:

```
genhalflogistic.pdf(x, c) =
    2 * (1-c*x)**(1/c-1) / (1+(1-c*x)**(1/c))**2
```

for `0 <= x <= 1/c`, and `c > 0`.

genhalflogistic takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `genhalflogistic.pdf(x, c, loc, scale)` is identically equivalent to `genhalflogistic.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import genhalflogistic
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.773
>>> mean, var, skew, kurt = genhalflogistic.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(genhalflogistic.ppf(0.01, c),
...                 genhalflogistic.ppf(0.99, c), 100)
>>> ax.plot(x, genhalflogistic.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='genhalflogistic pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = genhalflogistic(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

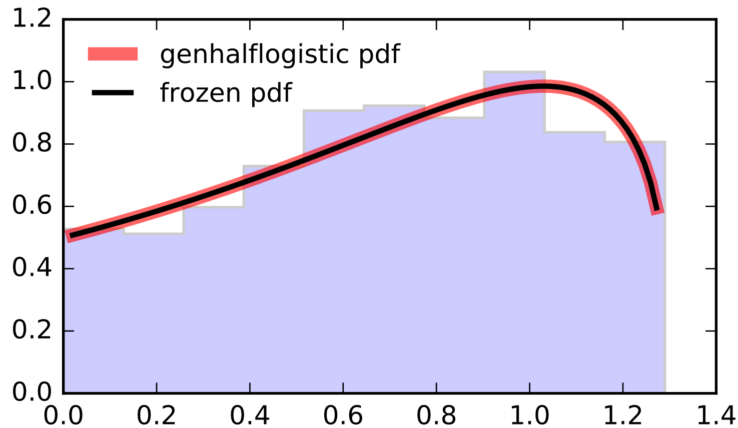
```
>>> vals = genhalflogistic.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], genhalflogistic.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = genhalflogistic.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`gilbrat`** **= <scipy.stats._continuous_distns.gilbrat_gen object>**

A Gilbrat continuous random variable.

As an instance of the *`rv_continuous`* class, *`gilbrat`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `gilbrat` is:

```
gilbrat.pdf(x) = 1/(x*sqrt(2*pi)) * exp(-1/2*(log(x))**2)
```

`gilbrat` is a special case of `lognorm` with `s = 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gilbrat.pdf(x, loc, scale)` is identically equivalent to `gilbrat.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gilbrat
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = gilbrat.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gilbrat.ppf(0.01),
...                 gilbrat.ppf(0.99), 100)
>>> ax.plot(x, gilbrat.pdf(x),
...        'r-', lw=5, alpha=0.6, label='gilbrat pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gilbrat()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

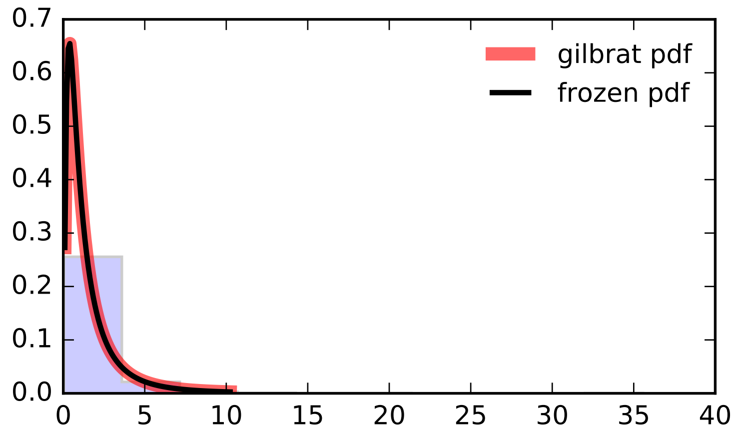
```
>>> vals = gilbrat.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], gilbrat.cdf(vals))
True
```

Generate random numbers:

```
>>> r = gilbrat.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`gompertz`** **= <scipy.stats._continuous_distns.gompertz_gen object>**

A Gompertz (or truncated Gumbel) continuous random variable.

As an instance of the *`rv_continuous`* class, *`gompertz`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `gompertz` is:

```
gompertz.pdf(x, c) = c * exp(x) * exp(-c*(exp(x)-1))
```

for `x >= 0`, `c > 0`.

`gompertz` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gompertz.pdf(x, c, loc, scale)` is identically equivalent to `gompertz.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gompertz
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.947
>>> mean, var, skew, kurt = gompertz.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gompertz.ppf(0.01, c),
...                 gompertz.ppf(0.99, c), 100)
>>> ax.plot(x, gompertz.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='gompertz pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gompertz(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

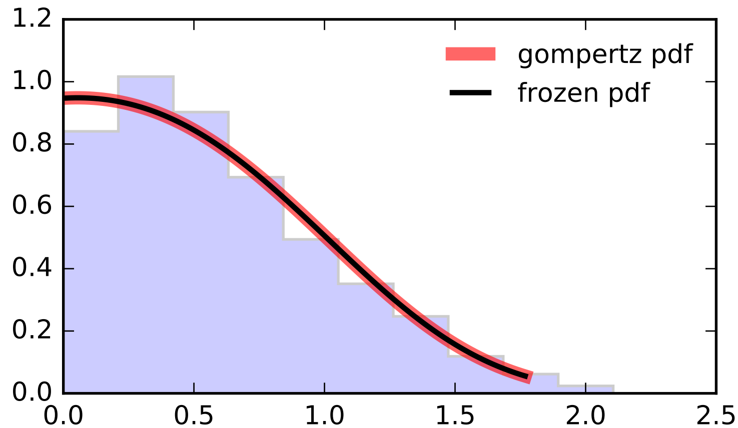
```
>>> vals = gompertz.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], gompertz.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = gompertz.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean(‘m’), variance(‘v’), skew(‘s’), and/or kurtosis(‘k’).
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`gumbel_r`** **= <scipy.stats._continuous_distns.gumbel_r_gen object>**

A right-skewed Gumbel continuous random variable.

As an instance of the *`rv_continuous`* class, *`gumbel_r`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

gumbel_l, *gompertz*, *genextreme*

Notes

The probability density function for `gumbel_r` is:

```
gumbel_r.pdf(x) = exp(-(x + exp(-x)))
```

The Gumbel distribution is sometimes referred to as a type I Fisher-Tippett distribution. It is also related to the extreme value distribution, log-Weibull and Gompertz distributions.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gumbel_r.pdf(x, loc, scale)` is identically equivalent to `gumbel_r.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gumbel_r
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = gumbel_r.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gumbel_r.ppf(0.01),
...                 gumbel_r.ppf(0.99), 100)
>>> ax.plot(x, gumbel_r.pdf(x),
...        'r-', lw=5, alpha=0.6, label='gumbel_r pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gumbel_r()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

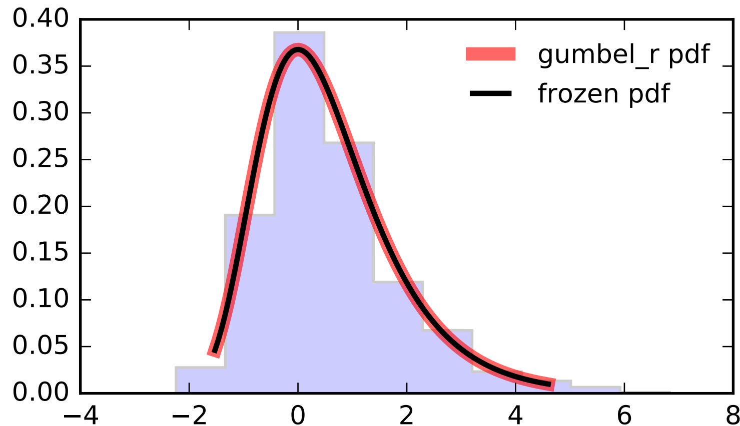
```
>>> vals = gumbel_r.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], gumbel_r.cdf(vals))
True
```

Generate random numbers:

```
>>> r = gumbel_r.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`gumbel_l`** **= <scipy.stats._continuous_distns.gumbel_l_gen object>**

A left-skewed Gumbel continuous random variable.

As an instance of the *`rv_continuous`* class, *`gumbel_l`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

gumbel_r, *gompertz*, *genextreme*

Notes

The probability density function for `gumbel_l` is:

```
gumbel_l.pdf(x) = exp(x - exp(x))
```

The Gumbel distribution is sometimes referred to as a type I Fisher-Tippett distribution. It is also related to the extreme value distribution, log-Weibull and Gompertz distributions.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gumbel_l.pdf(x, loc, scale)` is identically equivalent to `gumbel_l.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import gumbel_l
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = gumbel_l.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(gumbel_l.ppf(0.01),
...                 gumbel_l.ppf(0.99), 100)
>>> ax.plot(x, gumbel_l.pdf(x),
...        'r-', lw=5, alpha=0.6, label='gumbel_l pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = gumbel_l()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

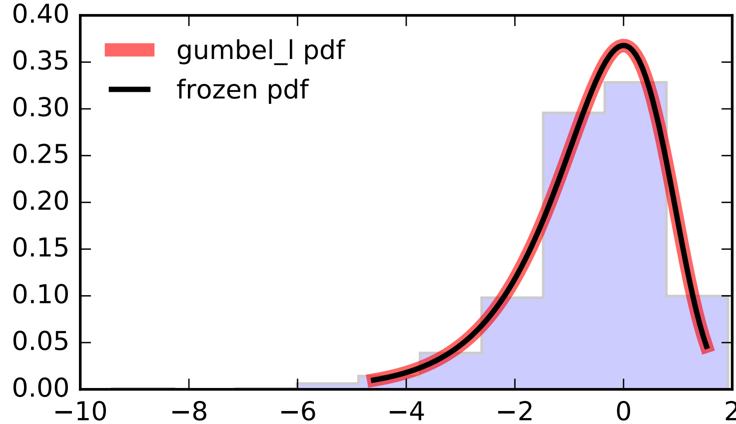
```
>>> vals = gumbel_l.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], gumbel_l.cdf(vals))
True
```

Generate random numbers:

```
>>> r = gumbel_l.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`halfcauchy`** **= <scipy.stats._continuous_distns.halfcauchy_gen object>**

A Half-Cauchy continuous random variable.

As an instance of the *`rv_continuous`* class, *`halfcauchy`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *halfcauchy* is:

```
halfcauchy.pdf(x) = 2 / (pi * (1 + x**2))
```

for `x >= 0`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `halfcauchy.pdf(x, loc, scale)` is identically equivalent to `halfcauchy.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import halfcauchy
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = halfcauchy.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(halfcauchy.ppf(0.01),
...                 halfcauchy.ppf(0.99), 100)
>>> ax.plot(x, halfcauchy.pdf(x),
...        'r-', lw=5, alpha=0.6, label='halfcauchy pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = halfcauchy()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

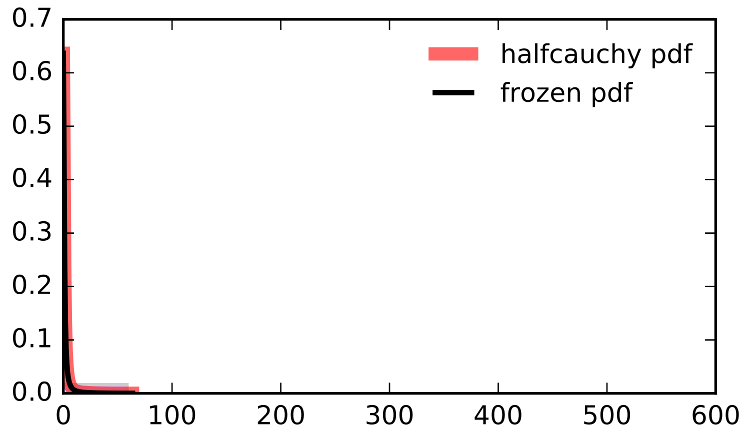
```
>>> vals = halfcauchy.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], halfcauchy.cdf(vals))
True
```

Generate random numbers:

```
>>> r = halfcauchy.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`halflogistic`** **= <scipy.stats._continuous_distns.halflogistic_gen object>**

A half-logistic continuous random variable.

As an instance of the *`rv_continuous`* class, *`halflogistic`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `halflogistic` is:

```
halflogistic.pdf(x) = 2 * exp(-x) / (1+exp(-x))**2 = 1/2 * sech(x/2)**2
```

for `x >= 0`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `halflogistic.pdf(x, loc, scale)` is identically equivalent to `halflogistic.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import halflogistic
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = halflogistic.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(halflogistic.ppf(0.01),
...                 halflogistic.ppf(0.99), 100)
>>> ax.plot(x, halflogistic.pdf(x),
...        'r-', lw=5, alpha=0.6, label='halflogistic pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = halflogistic()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

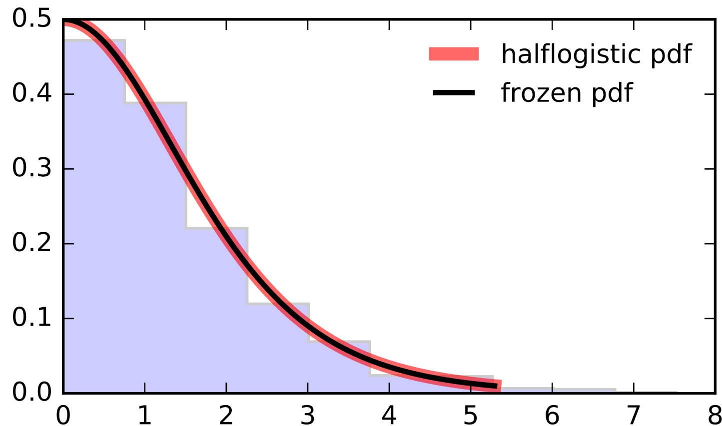
```
>>> vals = halflogistic.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], halflogistic.cdf(vals))
True
```

Generate random numbers:

```
>>> r = halflogistic.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`halfnorm`** **= <scipy.stats._continuous_distns.halfnorm_gen object>**

A half-normal continuous random variable.

As an instance of the *`rv_continuous`* class, *`halfnorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `halfnorm` is:

```
halfnorm.pdf(x) = sqrt(2/pi) * exp(-x**2/2)
```

for `x > 0`.

`halfnorm` is a special case of `chi` with `df == 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `halfnorm.pdf(x, loc, scale)` is identically equivalent to `halfnorm.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import halfnorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = halfnorm.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(halfnorm.ppf(0.01),
...                 halfnorm.ppf(0.99), 100)
>>> ax.plot(x, halfnorm.pdf(x),
...        'r-', lw=5, alpha=0.6, label='halfnorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = halfnorm()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

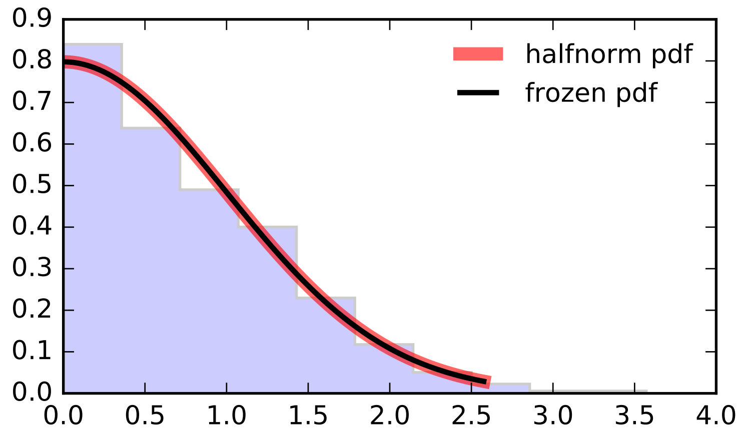
```
>>> vals = halfnorm.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], halfnorm.cdf(vals))
True
```

Generate random numbers:

```
>>> r = halfnorm.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`halfgennorm`** **= <scipy.stats._continuous_distns.halfgennorm_gen object>**

The upper half of a generalized normal continuous random variable.

As an instance of the *`rv_continuous`* class, *`halfgennorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

gennorm	generalized normal distribution
expon	exponential distribution
halfnorm	half normal distribution

Notes

The probability density function for *halfgennorm* is:

```
                                 beta
halfgennorm.pdf(x, beta) =  -------------  exp(-|x|**beta)
                            gamma(1/beta)
```

gennorm takes `beta` as a shape parameter. For `beta = 1`, it is identical to an exponential distribution. For `beta = 2`, it is identical to a half normal distribution (with `scale=1/sqrt(2)`).

References

[R588]

Examples

```
>>> from scipy.stats import halfgennorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> beta = 0.675
>>> mean, var, skew, kurt = halfgennorm.stats(beta, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(halfgennorm.ppf(0.01, beta),
...                 halfgennorm.ppf(0.99, beta), 100)
>>> ax.plot(x, halfgennorm.pdf(x, beta),
...        'r-', lw=5, alpha=0.6, label='halfgennorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = halfgennorm(beta)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

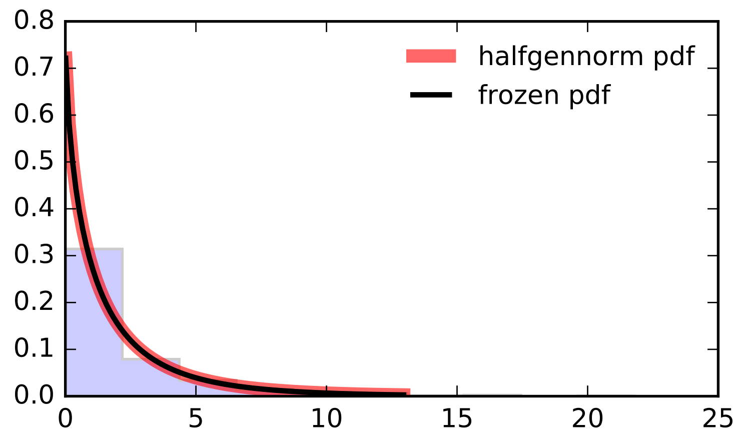
```
>>> vals = halfgennorm.ppf([0.001, 0.5, 0.999], beta)
>>> np.allclose([0.001, 0.5, 0.999], halfgennorm.cdf(vals, beta))
True
```

Generate random numbers:

```
>>> r = halfgennorm.rvs(beta, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(beta, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, beta, loc=0, scale=1)`	Probability density function.
`logpdf(x, beta, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, beta, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, beta, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, beta, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, beta, loc=0, scale=1)`	Log of the survival function.
`ppf(q, beta, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, beta, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, beta, loc=0, scale=1)`	Non-central moment of order n
`stats(beta, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(beta, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, beta, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(beta,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(beta, loc=0, scale=1)`	Median of the distribution.
`mean(beta, loc=0, scale=1)`	Mean of the distribution.
`var(beta, loc=0, scale=1)`	Variance of the distribution.
`std(beta, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, beta, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**hypsecant = <scipy.stats._continuous_distns.hypsecant_gen object>**

A hyperbolic secant continuous random variable.

As an instance of the *rv_continuous* class, *hypsecant* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *hypsecant* is:

```
hypsecant.pdf(x) = 1/pi * sech(x)
```

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `hypsecant.pdf(x, loc, scale)` is identically equivalent to `hypsecant.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import hypsecant
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = hypsecant.stats(moments='mvsk')
```

Display the probability density function (pdf):

```
>>> x = np.linspace(hypsecant.ppf(0.01),
...                 hypsecant.ppf(0.99), 100)
>>> ax.plot(x, hypsecant.pdf(x),
...        'r-', lw=5, alpha=0.6, label='hypsecant pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen pdf:

```
>>> rv = hypsecant()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of cdf and ppf:

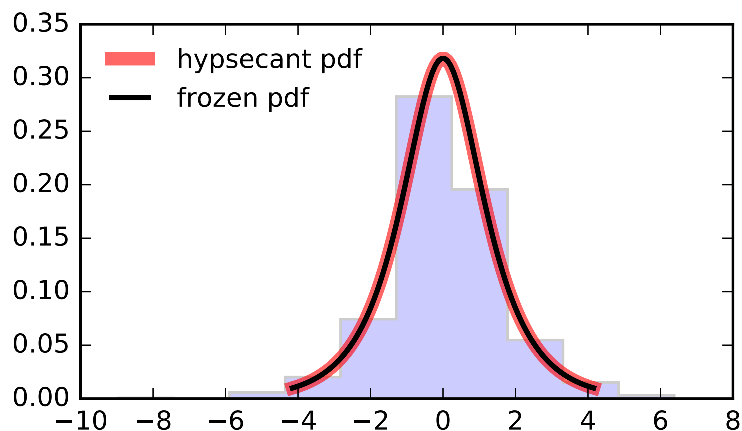
```
>>> vals = hypsecant.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], hypsecant.cdf(vals))
True
```

Generate random numbers:

```
>>> r = hypsecant.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**invgamma = <scipy.stats._continuous_distns.invgamma_gen object>**

An inverted gamma continuous random variable.

As an instance of the *rv_continuous* class, *invgamma* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *invgamma* is:

```
invgamma.pdf(x, a) = x**(-a-1) / gamma(a) * exp(-1/x)
```

for x > 0, a > 0.

invgamma takes `a` as a shape parameter.

invgamma is a special case of *gengamma* with `c == -1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `invgamma.pdf(x, a, loc, scale)` is identically equivalent to `invgamma.pdf(y, a) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import invgamma
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 4.07
>>> mean, var, skew, kurt = invgamma.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(invgamma.ppf(0.01, a),
...                 invgamma.ppf(0.99, a), 100)
>>> ax.plot(x, invgamma.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='invgamma pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = invgamma(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

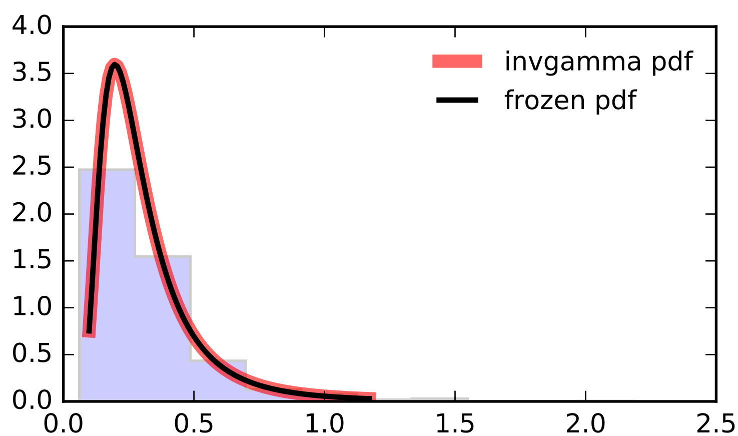
```
>>> vals = invgamma.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], invgamma.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = invgamma.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**invgauss = <scipy.stats._continuous_distns.invgauss_gen object>**

An inverse Gaussian continuous random variable.

As an instance of the *rv_continuous* class, *invgauss* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *invgauss* is:

```
invgauss.pdf(x, mu) = 1 / sqrt(2*pi*x**3) * exp(-(x-mu)**2/(2*x*mu**2))
```

for `x > 0`.

invgauss takes `mu` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `invgauss.pdf(x, mu, loc, scale)` is identically equivalent to `invgauss.pdf(y, mu) / scale` with `y = (x - loc) / scale`.

When *mu* is too small, evaluating the cumulative distribution function will be inaccurate due to `cdf(mu -> 0) = inf * 0`. NaNs are returned for `mu <= 0.0028`.

Examples

```
>>> from scipy.stats import invgauss
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mu = 0.145
>>> mean, var, skew, kurt = invgauss.stats(mu, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(invgauss.ppf(0.01, mu),
...                 invgauss.ppf(0.99, mu), 100)
>>> ax.plot(x, invgauss.pdf(x, mu),
...        'r-', lw=5, alpha=0.6, label='invgauss pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = invgauss(mu)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

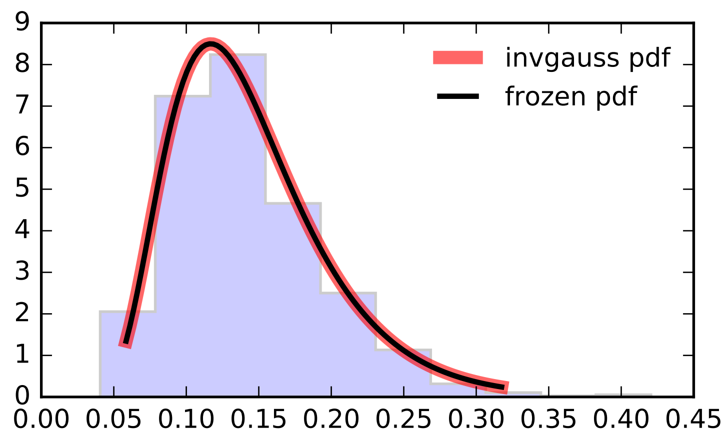
```
>>> vals = invgauss.ppf([0.001, 0.5, 0.999], mu)
>>> np.allclose([0.001, 0.5, 0.999], invgauss.cdf(vals, mu))
True
```

Generate random numbers:

```
>>> r = invgauss.rvs(mu, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(mu, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, mu, loc=0, scale=1)`	Probability density function.
`logpdf(x, mu, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, mu, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, mu, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, mu, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, mu, loc=0, scale=1)`	Log of the survival function.
`ppf(q, mu, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, mu, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, mu, loc=0, scale=1)`	Non-central moment of order n
`stats(mu, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(mu, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, mu, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(mu,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(mu, loc=0, scale=1)`	Median of the distribution.
`mean(mu, loc=0, scale=1)`	Mean of the distribution.
`var(mu, loc=0, scale=1)`	Variance of the distribution.
`std(mu, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, mu, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`invweibull`** **= <scipy.stats._continuous_distns.invweibull_gen object>**

An inverted Weibull continuous random variable.

As an instance of the *`rv_continuous`* class, *`invweibull`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *`invweibull`* is:

```
invweibull.pdf(x, c) = c * x**(-c-1) * exp(-x**(-c))
```

for `x > 0`, `c > 0`.

`invweibull` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `invweibull.pdf(x, c, loc, scale)` is identically equivalent to `invweibull.pdf(y, c) / scale` with `y = (x - loc) / scale`.

References

F.R.S. de Gusmao, E.M.M Ortega and G.M. Cordeiro, "The generalized inverse Weibull distribution", Stat. Papers, vol. 52, pp. 591-619, 2011.

Examples

```
>>> from scipy.stats import invweibull
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 10.6
>>> mean, var, skew, kurt = invweibull.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(invweibull.ppf(0.01, c),
...                 invweibull.ppf(0.99, c), 100)
>>> ax.plot(x, invweibull.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='invweibull pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = invweibull(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

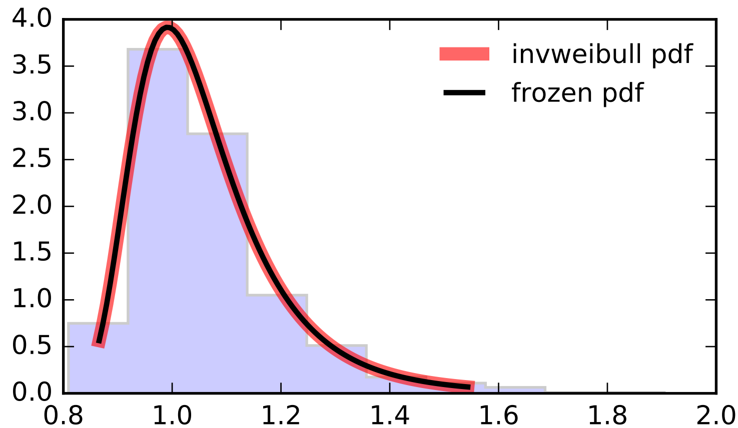
```
>>> vals = invweibull.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], invweibull.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = invweibull.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**johnsonsb = <scipy.stats._continuous_distns.johnsonsb_gen object>**

A Johnson SB continuous random variable.

As an instance of the *rv_continuous* class, *johnsonsb* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

johnsonsu

Notes

The probability density function for `johnsonsb` is:

```
johnsonsb.pdf(x, a, b) = b / (x*(1-x)) * phi(a + b * log(x/(1-x)))
```

for `0 < x < 1` and `a, b > 0`, and `phi` is the normal pdf.

`johnsonsb` takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `johnsonsb.pdf(x, a, b, loc, scale)` is identically equivalent to `johnsonsb.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import johnsonsb
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 4.32, 3.18
>>> mean, var, skew, kurt = johnsonsb.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(johnsonsb.ppf(0.01, a, b),
...                 johnsonsb.ppf(0.99, a, b), 100)
>>> ax.plot(x, johnsonsb.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='johnsonsb pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = johnsonsb(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

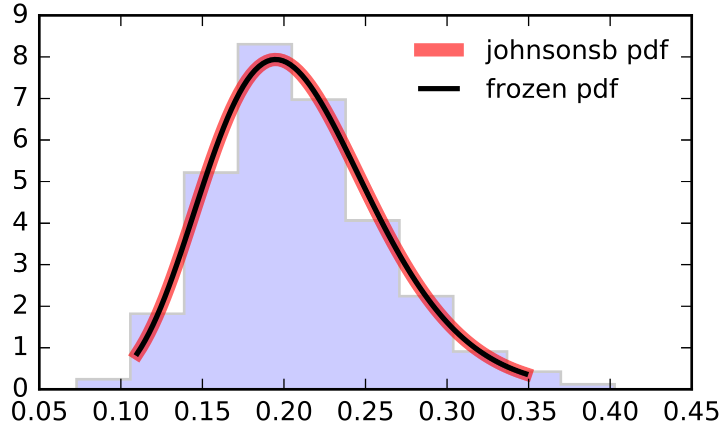
```
>>> vals = johnsonsb.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], johnsonsb.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = johnsonsb.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`johnsonsu`** **= <scipy.stats._continuous_distns.johnsonsu_gen object>**

A Johnson SU continuous random variable.

As an instance of the *`rv_continuous`* class, *`johnsonsu`* object inherits from it a collection of generic

methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

johnsonsb

Notes

The probability density function for `johnsonsu` is:

```
johnsonsu.pdf(x, a, b) = b / sqrt(x**2 + 1) *
                         phi(a + b * log(x + sqrt(x**2 + 1)))
```

for all `x, a, b > 0`, and *phi* is the normal pdf.

`johnsonsu` takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `johnsonsu.pdf(x, a, b, loc, scale)` is identically equivalent to `johnsonsu.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import johnsonsu
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 2.55, 2.25
>>> mean, var, skew, kurt = johnsonsu.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(johnsonsu.ppf(0.01, a, b),
...                 johnsonsu.ppf(0.99, a, b), 100)
>>> ax.plot(x, johnsonsu.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='johnsonsu pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = johnsonsu(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

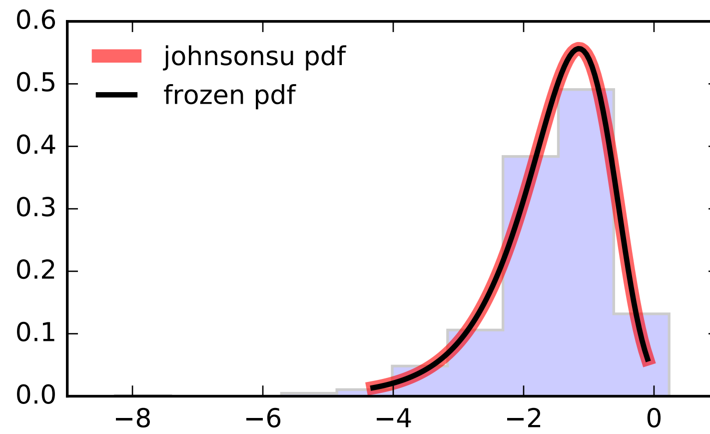
```
>>> vals = johnsonsu.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], johnsonsu.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = johnsonsu.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

rvs(a, b, loc=0, scale=1, size=1, random_state=None)	Random variates.
pdf(x, a, b, loc=0, scale=1)	Probability density function.
logpdf(x, a, b, loc=0, scale=1)	Log of the probability density function.
cdf(x, a, b, loc=0, scale=1)	Cumulative distribution function.
logcdf(x, a, b, loc=0, scale=1)	Log of the cumulative distribution function.
sf(x, a, b, loc=0, scale=1)	Survival function (also defined as 1 - cdf, but *sf* is sometimes more accurate).
logsf(x, a, b, loc=0, scale=1)	Log of the survival function.
ppf(q, a, b, loc=0, scale=1)	Percent point function (inverse of cdf — percentiles).
isf(q, a, b, loc=0, scale=1)	Inverse survival function (inverse of sf).
moment(n, a, b, loc=0, scale=1)	Non-central moment of order n
stats(a, b, loc=0, scale=1, moments='mv')	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
entropy(a, b, loc=0, scale=1)	(Differential) entropy of the RV.
fit(data, a, b, loc=0, scale=1)	Parameter estimates for generic data.
expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)	Expected value of a function (of one argument) with respect to the distribution.
median(a, b, loc=0, scale=1)	Median of the distribution.
mean(a, b, loc=0, scale=1)	Mean of the distribution.
var(a, b, loc=0, scale=1)	Variance of the distribution.
std(a, b, loc=0, scale=1)	Standard deviation of the distribution.
interval(alpha, a, b, loc=0, scale=1)	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**kappa4 = <scipy.stats._continuous_distns.kappa4_gen object>**

Kappa 4 parameter distribution.

As an instance of the *rv_continuous* class, *kappa4* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for kappa4 is:

```
kappa4.pdf(x, h, k) = (1.0 - k*x)**(1.0/k - 1.0)*
                      (1.0 - h*(1.0 - k*x)**(1.0/k))**(1.0/h-1)
```

if h and k are not equal to 0.

If h or k are zero then the pdf can be simplified:

h = 0 and k != 0:

```
kappa4.pdf(x, h, k) = (1.0 - k*x)**(1.0/k - 1.0)*
                      exp(-(1.0 - k*x)**(1.0/k))
```

h != 0 and k = 0:

```
kappa4.pdf(x, h, k) = exp(-x)*(1.0 - h*exp(-x))**(1.0/h - 1.0)
```

h = 0 and k = 0:

```
kappa4.pdf(x, h, k) = exp(-x)*exp(-exp(-x))
```

kappa4 takes `h` and `k` as shape parameters.

The kappa4 distribution returns other distributions when certain `h` and `k` values are used.

h	k=0.0	k=1.0	-inf<=k<=inf
-1.0	Logistic logistic(x)		Generalized Logistic(1)
0.0	Gumbel gumbel_r(x)	Reverse Exponential(2)	Generalized Extreme Value genextreme(x, k)
1.0	Exponential expon(x)	Uniform uniform(x)	Generalized Pareto genpareto(x, -k)

1. There are at least five generalized logistic distributions. Four are described here: https://en.wikipedia.org/wiki/Generalized_logistic_distribution The "fifth" one is the one kappa4 should match which currently isn't implemented in scipy: https://en.wikipedia.org/wiki/Talk:Generalized_logistic_distribution http://www.mathwave.com/help/easyfit/html/analyses/distributions/gen_logistic.html
2. This distribution is currently not in scipy.

References

J.C. Finney, "Optimization of a Skewed Logistic Distribution With Respect to the Kolmogorov-Smirnov Test", A Dissertation Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College, (August, 2004), http://etd.lsu.edu/docs/available/etd-05182004-144851/unrestricted/Finney_dis.pdf

J.R.M. Hosking, "The four-parameter kappa distribution". IBM J. Res. Develop. 38 (3), 25 1-258 (1994).

B. Kumphon, A. Kaew-Man, P. Seenoi, "A Rainfall Distribution for the Lampao Site in the Chi River Basin, Thailand", Journal of Water Resource and Protection, vol. 4, 866-869, (2012). http://file.scirp.org/pdf/JWARP20121000009_14676002.pdf

C. Winchester, "On Estimation of the Four-Parameter Kappa Distribution", A Thesis Submitted to Dalhousie University, Halifax, Nova Scotia, (March 2000). http://www.nlc-bnc.ca/obj/s4/f2/dsk2/ftp01/MQ57336.pdf

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `kappa4.pdf(x, h, k, loc, scale)` is identically equivalent to `kappa4.pdf(y, h, k) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import kappa4
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> h, k = 0.1, 0
>>> mean, var, skew, kurt = kappa4.stats(h, k, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(kappa4.ppf(0.01, h, k),
...                 kappa4.ppf(0.99, h, k), 100)
>>> ax.plot(x, kappa4.pdf(x, h, k),
...        'r-', lw=5, alpha=0.6, label='kappa4 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen pdf:

```
>>> rv = kappa4(h, k)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of cdf and ppf:

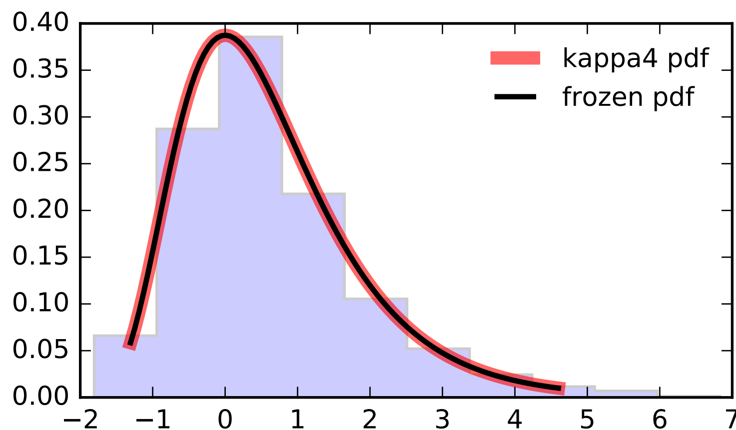
```
>>> vals = kappa4.ppf([0.001, 0.5, 0.999], h, k)
>>> np.allclose([0.001, 0.5, 0.999], kappa4.cdf(vals, h, k))
True
```

Generate random numbers:

```
>>> r = kappa4.rvs(h, k, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(h, k, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, h, k, loc=0, scale=1)`	Probability density function.
`logpdf(x, h, k, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, h, k, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, h, k, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, h, k, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, h, k, loc=0, scale=1)`	Log of the survival function.
`ppf(q, h, k, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, h, k, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, h, k, loc=0, scale=1)`	Non-central moment of order n
`stats(h, k, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(h, k, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, h, k, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(h, k), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(h, k, loc=0, scale=1)`	Median of the distribution.
`mean(h, k, loc=0, scale=1)`	Mean of the distribution.
`var(h, k, loc=0, scale=1)`	Variance of the distribution.
`std(h, k, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, h, k, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**kappa3 = <scipy.stats._continuous_distns.kappa3_gen object>**

Kappa 3 parameter distribution.

As an instance of the *rv_continuous* class, *kappa3* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *kappa* is:

```
kappa3.pdf(x, a) =
    a*[a + x**a]**(-(a + 1)/a),     for ``x > 0``
    0.0,                            for ``x <= 0``
```

kappa3 takes `a` as a shape parameter and `a > 0`.

References

P.W. Mielke and E.S. Johnson, "Three-Parameter Kappa Distribution Maximum Likelihood and Likelihood Ratio Tests", Methods in Weather Research, 701-707, (September, 1973), http://docs.lib.noaa.gov/rescue/mwr/101/mwr-101-09-0701.pdf

B. Kumphon, "Maximum Entropy and Maximum Likelihood Estimation for the Three-Parameter Kappa Distribution", Open Journal of Statistics, vol 2, 415-419 (2012) http://file.scirp.org/pdf/OJS20120400011_95789012.pdf

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `kappa3.pdf(x, a, loc, scale)` is identically equivalent to `kappa3.pdf(y, a) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import kappa3
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 1
>>> mean, var, skew, kurt = kappa3.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(kappa3.ppf(0.01, a),
...                 kappa3.ppf(0.99, a), 100)
>>> ax.plot(x, kappa3.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='kappa3 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = kappa3(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

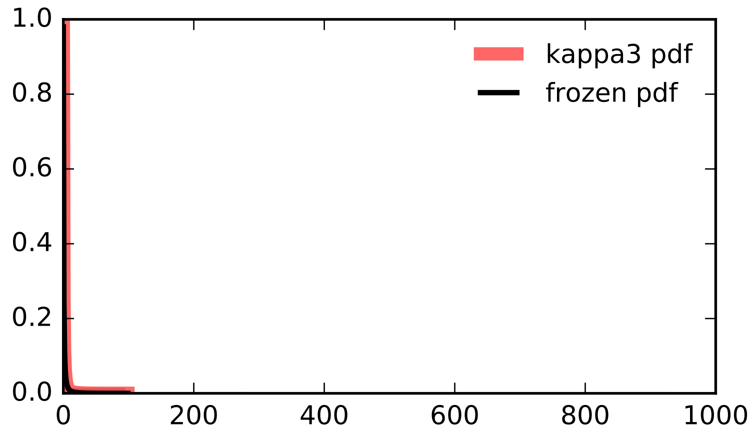
```
>>> vals = kappa3.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], kappa3.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = kappa3.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**ksone = <scipy.stats._continuous_distns.ksone_gen object>**
General Kolmogorov-Smirnov one-sided test.

As an instance of the *rv_continuous* class, *ksone* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Examples

```
>>> from scipy.stats import ksone
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> n = 1e+03
>>> mean, var, skew, kurt = ksone.stats(n, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(ksone.ppf(0.01, n),
...                 ksone.ppf(0.99, n), 100)
>>> ax.plot(x, ksone.pdf(x, n),
...        'r-', lw=5, alpha=0.6, label='ksone pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = ksone(n)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

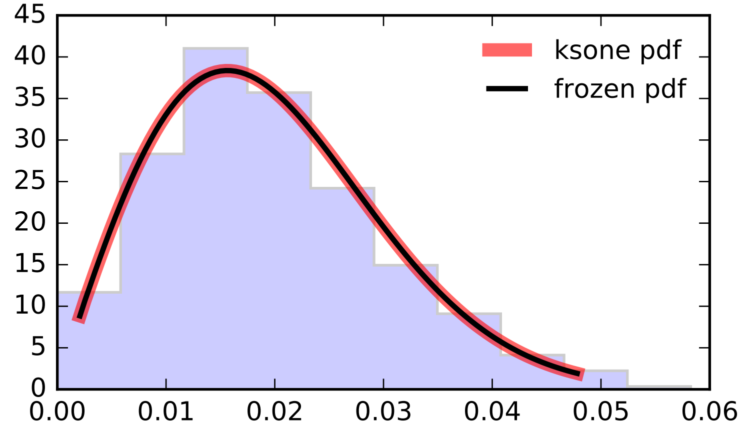
```
>>> vals = ksone.ppf([0.001, 0.5, 0.999], n)
>>> np.allclose([0.001, 0.5, 0.999], ksone.cdf(vals, n))
True
```

Generate random numbers:

```
>>> r = ksone.rvs(n, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(n, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, n, loc=0, scale=1)`	Probability density function.
`logpdf(x, n, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, n, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, n, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, n, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, n, loc=0, scale=1)`	Log of the survival function.
`ppf(q, n, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, n, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, n, loc=0, scale=1)`	Non-central moment of order n
`stats(n, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(n, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, n, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(n,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(n, loc=0, scale=1)`	Median of the distribution.
`mean(n, loc=0, scale=1)`	Mean of the distribution.
`var(n, loc=0, scale=1)`	Variance of the distribution.
`std(n, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, n, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`kstwobign`** **= <scipy.stats._continuous_distns.kstwobign_gen object>**
Kolmogorov-Smirnov two-sided test for large N.

As an instance of the *`rv_continuous`* class, *`kstwobign`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Examples

```
>>> from scipy.stats import kstwobign
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = kstwobign.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(kstwobign.ppf(0.01),
...                 kstwobign.ppf(0.99), 100)
>>> ax.plot(x, kstwobign.pdf(x),
...        'r-', lw=5, alpha=0.6, label='kstwobign pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = kstwobign()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

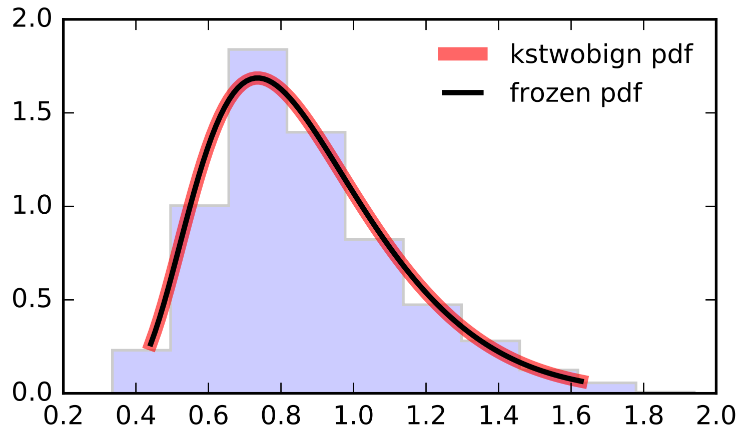
```
>>> vals = kstwobign.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], kstwobign.cdf(vals))
True
```

Generate random numbers:

```
>>> r = kstwobign.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`laplace`** **= <scipy.stats._continuous_distns.laplace_gen object>**

A Laplace continuous random variable.

As an instance of the *`rv_continuous`* class, *`laplace`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `laplace` is:

```
laplace.pdf(x) = 1/2 * exp(-abs(x))
```

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `laplace.pdf(x, loc, scale)` is identically equivalent to `laplace.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import laplace
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = laplace.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(laplace.ppf(0.01),
...                 laplace.ppf(0.99), 100)
>>> ax.plot(x, laplace.pdf(x),
...        'r-', lw=5, alpha=0.6, label='laplace pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = laplace()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

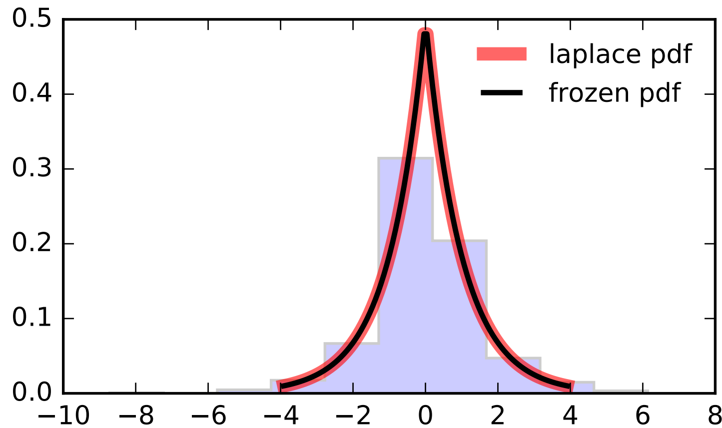
```
>>> vals = laplace.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], laplace.cdf(vals))
True
```

Generate random numbers:

```
>>> r = laplace.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`levy`** **= <scipy.stats._continuous_distns.levy_gen object>**

A Levy continuous random variable.

As an instance of the *`rv_continuous`* class, *`levy`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

levy_stable, *levy_l*

Notes

The probability density function for `levy` is:

```
levy.pdf(x) = 1 / (x * sqrt(2*pi*x)) * exp(-1/(2*x))
```

for `x > 0`.

This is the same as the Levy-stable distribution with a=1/2 and b=1.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `levy.pdf(x, loc, scale)` is identically equivalent to `levy.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import levy
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = levy.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(levy.ppf(0.01),
...                 levy.ppf(0.99), 100)
>>> ax.plot(x, levy.pdf(x),
...        'r-', lw=5, alpha=0.6, label='levy pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = levy()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

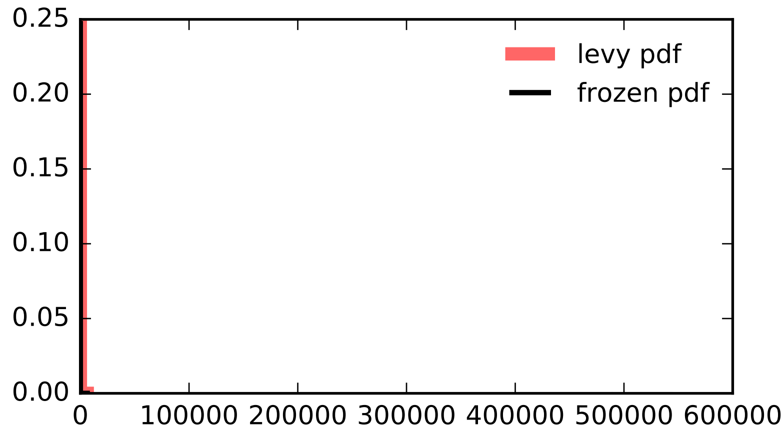
```
>>> vals = levy.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], levy.cdf(vals))
True
```

Generate random numbers:

```
>>> r = levy.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`levy_l`** **= <scipy.stats._continuous_distns.levy_l_gen object>**

A left-skewed Levy continuous random variable.

As an instance of the *rv_continuous* class, *levy_l* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

levy, *levy_stable*

Notes

The probability density function for `levy_l` is:

```
levy_l.pdf(x) = 1 / (abs(x) * sqrt(2*pi*abs(x))) * exp(-1/(2*abs(x)))
```

for `x < 0`.

This is the same as the Levy-stable distribution with a=1/2 and b=-1.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `levy_l.pdf(x, loc, scale)` is identically equivalent to `levy_l.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import levy_l
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = levy_l.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(levy_l.ppf(0.01),
...                 levy_l.ppf(0.99), 100)
>>> ax.plot(x, levy_l.pdf(x),
...        'r-', lw=5, alpha=0.6, label='levy_l pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = levy_l()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

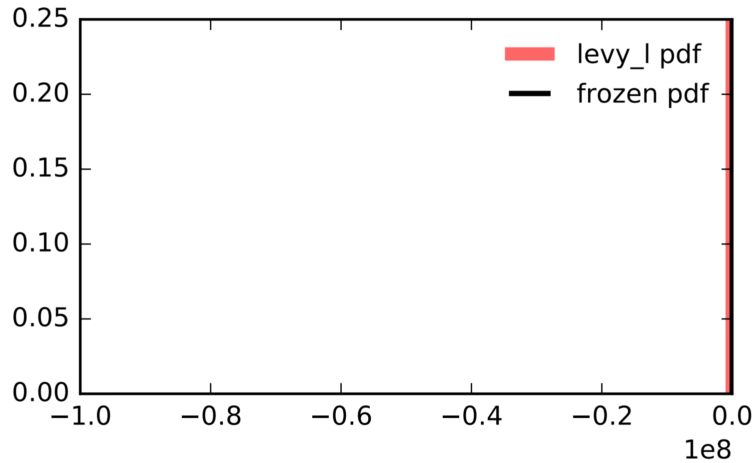
```
>>> vals = levy_l.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], levy_l.cdf(vals))
True
```

Generate random numbers:

```
>>> r = levy_l.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`levy_stable`** **= <scipy.stats._continuous_distns.levy_stable_gen object>**

A Levy-stable continuous random variable.

As an instance of the *`rv_continuous`* class, *`levy_stable`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

`levy`, `levy_l`

Notes

Levy-stable distribution (only random variates available – ignore other docs)

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `levy_stable.pdf(x, alpha, beta, loc, scale)` is identically equivalent to `levy_stable.pdf(y, alpha, beta) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import levy_stable
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> alpha, beta = 0.357, -0.675
>>> mean, var, skew, kurt = levy_stable.stats(alpha, beta, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(levy_stable.ppf(0.01, alpha, beta),
...                 levy_stable.ppf(0.99, alpha, beta), 100)
>>> ax.plot(x, levy_stable.pdf(x, alpha, beta),
...        'r-', lw=5, alpha=0.6, label='levy_stable pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = levy_stable(alpha, beta)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

```
>>> vals = levy_stable.ppf([0.001, 0.5, 0.999], alpha, beta)
>>> np.allclose([0.001, 0.5, 0.999], levy_stable.cdf(vals, alpha, beta))
True
```

Generate random numbers:

```
>>> r = levy_stable.rvs(alpha, beta, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```

Methods

`rvs(alpha, beta, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, alpha, beta, loc=0, scale=1)`	Probability density function.
`logpdf(x, alpha, beta, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, alpha, beta, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, alpha, beta, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, alpha, beta, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, alpha, beta, loc=0, scale=1)`	Log of the survival function.
`ppf(q, alpha, beta, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, alpha, beta, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, alpha, beta, loc=0, scale=1)`	Non-central moment of order n
`stats(alpha, beta, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(alpha, beta, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, alpha, beta, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(alpha, beta), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(alpha, beta, loc=0, scale=1)`	Median of the distribution.
`mean(alpha, beta, loc=0, scale=1)`	Mean of the distribution.
`var(alpha, beta, loc=0, scale=1)`	Variance of the distribution.
`std(alpha, beta, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, alpha, beta, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`logistic`** **= <scipy.stats._continuous_distns.logistic_gen object>**

A logistic (or Sech-squared) continuous random variable.

As an instance of the *rv_continuous* class, *logistic* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *logistic* is:

```
logistic.pdf(x) = exp(-x) / (1+exp(-x))**2
```

logistic is a special case of *genlogistic* with `c == 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `logistic.pdf(x, loc, scale)` is identically equivalent to `logistic.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import logistic
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = logistic.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(logistic.ppf(0.01),
...                 logistic.ppf(0.99), 100)
>>> ax.plot(x, logistic.pdf(x),
...        'r-', lw=5, alpha=0.6, label='logistic pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = logistic()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

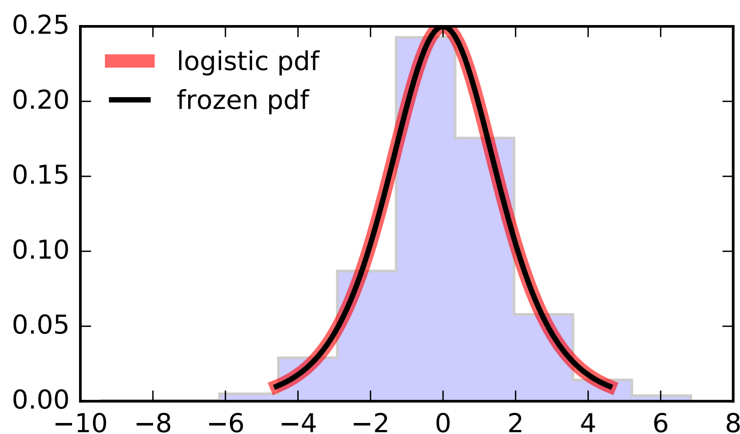
```
>>> vals = logistic.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], logistic.cdf(vals))
True
```

Generate random numbers:

```
>>> r = logistic.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**loggamma = <scipy.stats._continuous_distns.loggamma_gen object>**

A log gamma continuous random variable.

As an instance of the *rv_continuous* class, *loggamma* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *loggamma* is:

```
loggamma.pdf(x, c) = exp(c*x-exp(x)) / gamma(c)
```

for all `x, c > 0`.

loggamma takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `loggamma.pdf(x, c, loc, scale)` is identically equivalent to `loggamma.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import loggamma
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.414
>>> mean, var, skew, kurt = loggamma.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(loggamma.ppf(0.01, c),
...                 loggamma.ppf(0.99, c), 100)
>>> ax.plot(x, loggamma.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='loggamma pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = loggamma(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

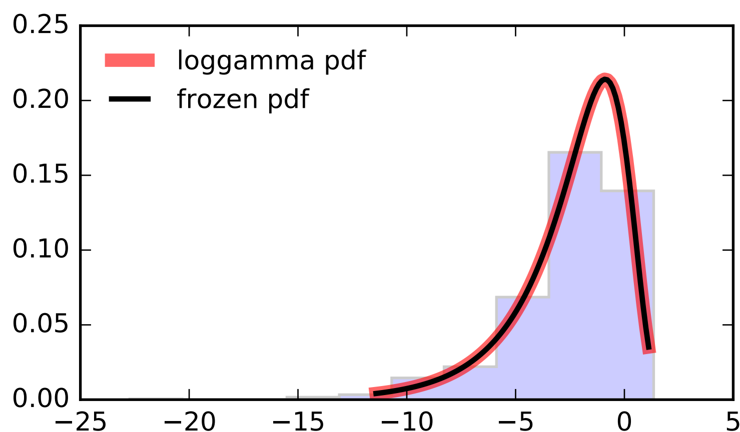
```
>>> vals = loggamma.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], loggamma.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = loggamma.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`loglaplace`** **= <scipy.stats._continuous_distns.loglaplace_gen object>**

A log-Laplace continuous random variable.

As an instance of the *`rv_continuous`* class, *`loglaplace`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *`loglaplace`* is:

```
loglaplace.pdf(x, c) = c / 2 * x**(c-1),   for 0 < x < 1
                     = c / 2 * x**(-c-1),  for x >= 1
```

for `c > 0`.

`loglaplace` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `loglaplace.pdf(x, c, loc, scale)` is identically equivalent to `loglaplace.pdf(y, c) / scale` with `y = (x - loc) / scale`.

References

T.J. Kozubowski and K. Podgorski, "A log-Laplace growth rate model", The Mathematical Scientist, vol. 28, pp. 49-60, 2003.

Examples

```
>>> from scipy.stats import loglaplace
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 3.25
>>> mean, var, skew, kurt = loglaplace.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(loglaplace.ppf(0.01, c),
...                 loglaplace.ppf(0.99, c), 100)
>>> ax.plot(x, loglaplace.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='loglaplace pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = loglaplace(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

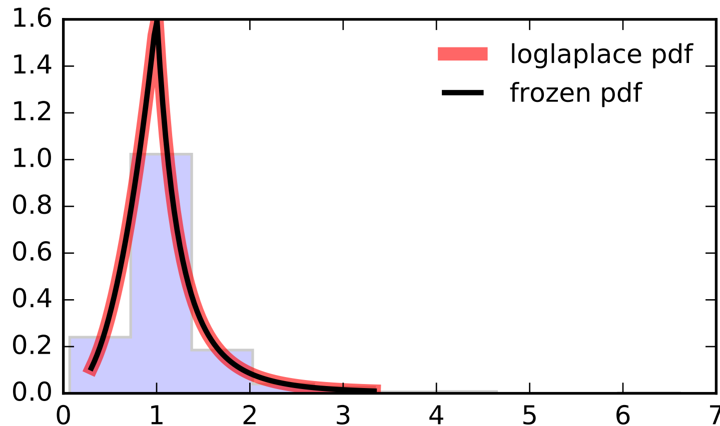
```
>>> vals = loglaplace.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], loglaplace.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = loglaplace.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`lognorm`** **= <scipy.stats._continuous_distns.lognorm_gen object>**

A lognormal continuous random variable.

As an instance of the *`rv_continuous`* class, *`lognorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `lognorm` is:

```
lognorm.pdf(x, s) = 1 / (s*x*sqrt(2*pi)) * exp(-1/2*(log(x)/s)**2)
```

for `x > 0`, `s > 0`.

`lognorm` takes `s` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `lognorm.pdf(x, s, loc, scale)` is identically equivalent to `lognorm.pdf(y, s) / scale` with `y = (x - loc) / scale`.

A common parametrization for a lognormal random variable `Y` is in terms of the mean, `mu`, and standard deviation, `sigma`, of the unique normally distributed random variable `X` such that exp(X) = Y. This parametrization corresponds to setting `s = sigma` and `scale = exp(mu)`.

Examples

```
>>> from scipy.stats import lognorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> s = 0.954
>>> mean, var, skew, kurt = lognorm.stats(s, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(lognorm.ppf(0.01, s),
...                 lognorm.ppf(0.99, s), 100)
>>> ax.plot(x, lognorm.pdf(x, s),
...        'r-', lw=5, alpha=0.6, label='lognorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = lognorm(s)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

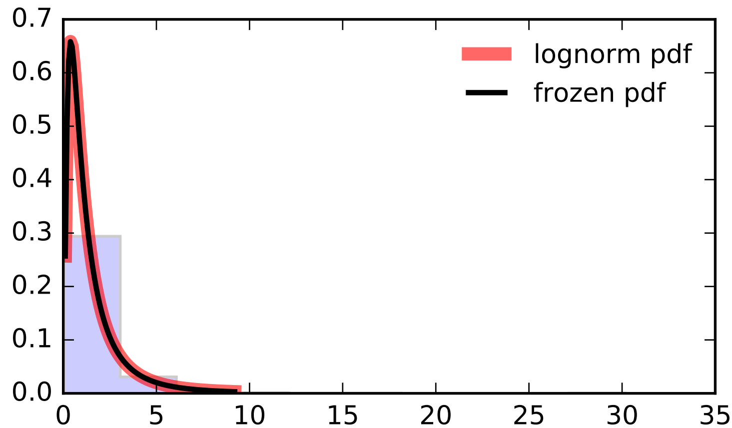
```
>>> vals = lognorm.ppf([0.001, 0.5, 0.999], s)
>>> np.allclose([0.001, 0.5, 0.999], lognorm.cdf(vals, s))
True
```

Generate random numbers:

```
>>> r = lognorm.rvs(s, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(s, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, s, loc=0, scale=1)`	Probability density function.
`logpdf(x, s, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, s, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, s, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, s, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, s, loc=0, scale=1)`	Log of the survival function.
`ppf(q, s, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, s, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, s, loc=0, scale=1)`	Non-central moment of order n
`stats(s, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(s, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, s, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(s,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(s, loc=0, scale=1)`	Median of the distribution.
`mean(s, loc=0, scale=1)`	Mean of the distribution.
`var(s, loc=0, scale=1)`	Variance of the distribution.
`std(s, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, s, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`lomax`** **= <scipy.stats._continuous_distns.lomax_gen object>**

A Lomax (Pareto of the second kind) continuous random variable.

As an instance of the *`rv_continuous`* class, *`lomax`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The Lomax distribution is a special case of the Pareto distribution, with (loc=-1.0).

The probability density function for *lomax* is:

```
lomax.pdf(x, c) = c / (1+x)**(c+1)
```

for `x >= 0`, `c > 0`.

lomax takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `lomax.pdf(x, c, loc, scale)` is identically equivalent to `lomax.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import lomax
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 1.88
>>> mean, var, skew, kurt = lomax.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(lomax.ppf(0.01, c),
...                 lomax.ppf(0.99, c), 100)
>>> ax.plot(x, lomax.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='lomax pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = lomax(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

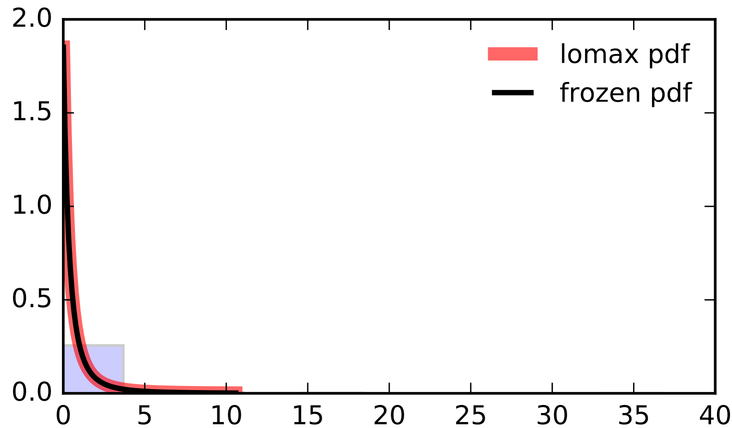
```
>>> vals = lomax.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], lomax.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = lomax.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`maxwell`** **= <scipy.stats._continuous_distns.maxwell_gen object>**
A Maxwell continuous random variable.

As an instance of the *`rv_continuous`* class, *`maxwell`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

A special case of a *chi* distribution, with `df = 3`, `loc = 0.0`, and given `scale = a`, where `a` is the parameter used in the Mathworld description *[R606]*.

The probability density function for *maxwell* is:

```
maxwell.pdf(x) = sqrt(2/pi)x**2 * exp(-x**2/2)
```

for `x > 0`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `maxwell.pdf(x, loc, scale)` is identically equivalent to `maxwell.pdf(y) / scale` with `y = (x - loc) / scale`.

References

[R606]

Examples

```
>>> from scipy.stats import maxwell
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = maxwell.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(maxwell.ppf(0.01),
...                 maxwell.ppf(0.99), 100)
>>> ax.plot(x, maxwell.pdf(x),
...        'r-', lw=5, alpha=0.6, label='maxwell pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = maxwell()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

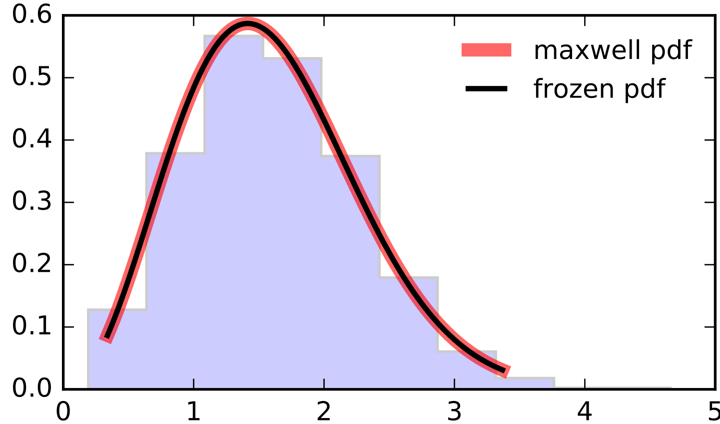
```
>>> vals = maxwell.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], maxwell.cdf(vals))
True
```

Generate random numbers:

```
>>> r = maxwell.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`mielke`** **= <scipy.stats._continuous_distns.mielke_gen object>**

A Mielke's Beta-Kappa continuous random variable.

As an instance of the *`rv_continuous`* class, *`mielke`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *mielke* is:

```
mielke.pdf(x, k, s) = k * x**(k-1) / (1+x**s)**(1+k/s)
```

for `x > 0`.

mielke takes `k` and `s` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `mielke.pdf(x, k, s, loc, scale)` is identically equivalent to `mielke.pdf(y, k, s) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import mielke
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> k, s = 10.4, 3.6
>>> mean, var, skew, kurt = mielke.stats(k, s, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(mielke.ppf(0.01, k, s),
...                 mielke.ppf(0.99, k, s), 100)
>>> ax.plot(x, mielke.pdf(x, k, s),
...        'r-', lw=5, alpha=0.6, label='mielke pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = mielke(k, s)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

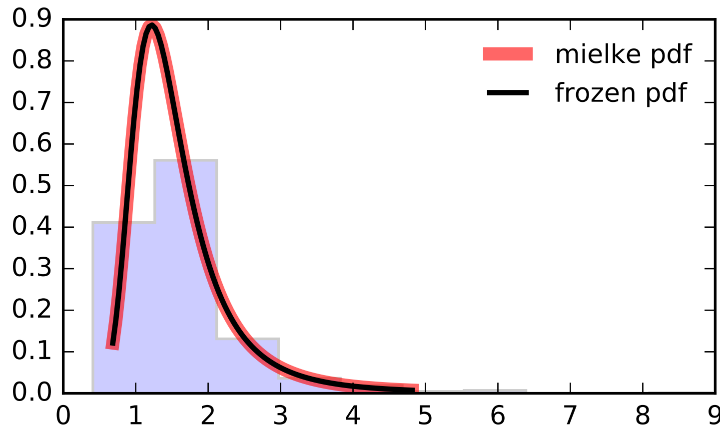
```
>>> vals = mielke.ppf([0.001, 0.5, 0.999], k, s)
>>> np.allclose([0.001, 0.5, 0.999], mielke.cdf(vals, k, s))
True
```

Generate random numbers:

```
>>> r = mielke.rvs(k, s, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(k, s, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, k, s, loc=0, scale=1)`	Probability density function.
`logpdf(x, k, s, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, k, s, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, k, s, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, k, s, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, k, s, loc=0, scale=1)`	Log of the survival function.
`ppf(q, k, s, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, k, s, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, k, s, loc=0, scale=1)`	Non-central moment of order n
`stats(k, s, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(k, s, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, k, s, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(k, s), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(k, s, loc=0, scale=1)`	Median of the distribution.
`mean(k, s, loc=0, scale=1)`	Mean of the distribution.
`var(k, s, loc=0, scale=1)`	Variance of the distribution.
`std(k, s, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, k, s, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`nakagami`** **= <scipy.stats._continuous_distns.nakagami_gen object>**

A Nakagami continuous random variable.

As an instance of the *`rv_continuous`* class, *`nakagami`* object inherits from it a collection of generic meth-

ods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `nakagami` is:

```
nakagami.pdf(x, nu) = 2 * nu**nu / gamma(nu) *
                      x**(2*nu-1) * exp(-nu*x**2)
```

for `x > 0`, `nu > 0`.

`nakagami` takes `nu` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `nakagami.pdf(x, nu, loc, scale)` is identically equivalent to `nakagami.pdf(y, nu) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import nakagami
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> nu = 4.97
>>> mean, var, skew, kurt = nakagami.stats(nu, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(nakagami.ppf(0.01, nu),
...                 nakagami.ppf(0.99, nu), 100)
>>> ax.plot(x, nakagami.pdf(x, nu),
...        'r-', lw=5, alpha=0.6, label='nakagami pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = nakagami(nu)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

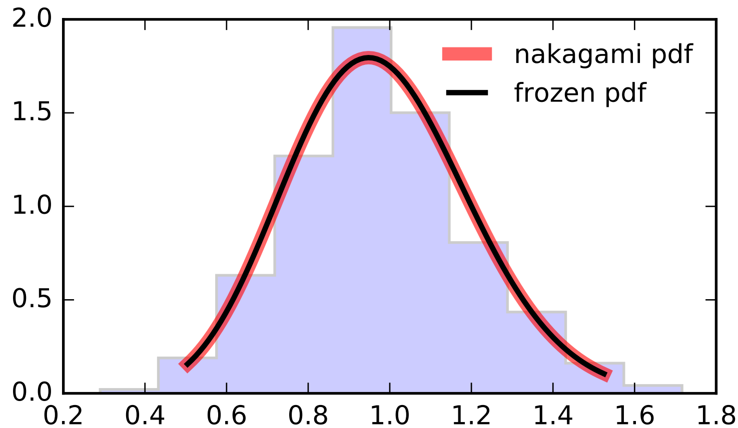
```
>>> vals = nakagami.ppf([0.001, 0.5, 0.999], nu)
>>> np.allclose([0.001, 0.5, 0.999], nakagami.cdf(vals, nu))
True
```

Generate random numbers:

```
>>> r = nakagami.rvs(nu, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(nu, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, nu, loc=0, scale=1)`	Probability density function.
`logpdf(x, nu, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, nu, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, nu, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, nu, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, nu, loc=0, scale=1)`	Log of the survival function.
`ppf(q, nu, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, nu, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, nu, loc=0, scale=1)`	Non-central moment of order n
`stats(nu, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(nu, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, nu, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(nu,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(nu, loc=0, scale=1)`	Median of the distribution.
`mean(nu, loc=0, scale=1)`	Mean of the distribution.
`var(nu, loc=0, scale=1)`	Variance of the distribution.
`std(nu, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, nu, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**ncx2 = <scipy.stats._continuous_distns.ncx2_gen object>**

A non-central chi-squared continuous random variable.

As an instance of the *rv_continuous* class, *ncx2* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `ncx2` is:

```
ncx2.pdf(x, df, nc) = exp(-(nc+x)/2) * 1/2 * (x/nc)**((df-2)/4)
                        * I[(df-2)/2](sqrt(nc*x))
```

for `x > 0`.

`ncx2` takes `df` and `nc` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `ncx2.pdf(x, df, nc, loc, scale)` is identically equivalent to `ncx2.pdf(y, df, nc) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import ncx2
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> df, nc = 21, 1.06
>>> mean, var, skew, kurt = ncx2.stats(df, nc, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(ncx2.ppf(0.01, df, nc),
...                 ncx2.ppf(0.99, df, nc), 100)
>>> ax.plot(x, ncx2.pdf(x, df, nc),
...        'r-', lw=5, alpha=0.6, label='ncx2 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = ncx2(df, nc)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

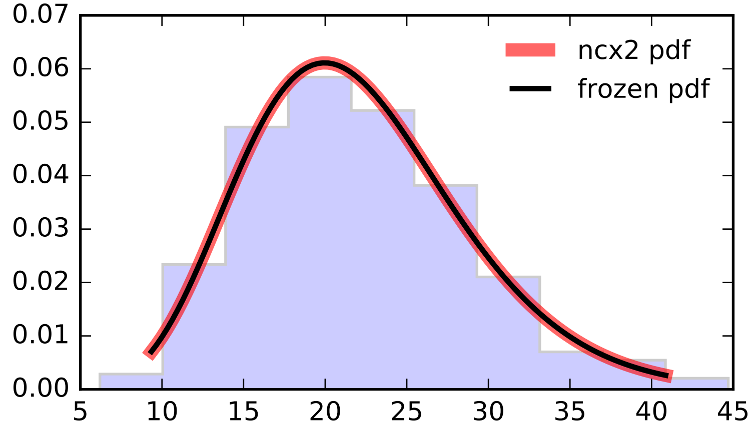
```
>>> vals = ncx2.ppf([0.001, 0.5, 0.999], df, nc)
>>> np.allclose([0.001, 0.5, 0.999], ncx2.cdf(vals, df, nc))
True
```

Generate random numbers:

```
>>> r = ncx2.rvs(df, nc, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(df, nc, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, df, nc, loc=0, scale=1)`	Probability density function.
`logpdf(x, df, nc, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, df, nc, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, df, nc, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, df, nc, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, df, nc, loc=0, scale=1)`	Log of the survival function.
`ppf(q, df, nc, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, df, nc, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, df, nc, loc=0, scale=1)`	Non-central moment of order n
`stats(df, nc, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(df, nc, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, df, nc, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(df, nc), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(df, nc, loc=0, scale=1)`	Median of the distribution.
`mean(df, nc, loc=0, scale=1)`	Mean of the distribution.
`var(df, nc, loc=0, scale=1)`	Variance of the distribution.
`std(df, nc, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, df, nc, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`ncf`** **= <scipy.stats._continuous_distns.ncf_gen object>**

A non-central F distribution continuous random variable.

As an instance of the *`rv_continuous`* class, *`ncf`* object inherits from it a collection of generic methods (see

below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `ncf` is:

```
ncf.pdf(x, df1, df2, nc) = exp(nc/2 + nc*df1*x/(2*(df1*x+df2))) *
            df1**(df1/2) * df2**(df2/2) * x**(df1/2-1) *
            (df2+df1*x)**(-(df1+df2)/2) *
            gamma(df1/2)*gamma(1+df2/2) *
            L^{v1/2-1}^{v2/2}(-nc*v1*x/(2*(v1*x+v2))) /
            (B(v1/2, v2/2) * gamma((v1+v2)/2))
```

for `df1, df2, nc > 0`.

`ncf` takes `df1`, `df2` and `nc` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `ncf.pdf(x, dfn, dfd, nc, loc, scale)` is identically equivalent to `ncf.pdf(y, dfn, dfd, nc) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import ncf
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> dfn, dfd, nc = 27, 27, 0.416
>>> mean, var, skew, kurt = ncf.stats(dfn, dfd, nc, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(ncf.ppf(0.01, dfn, dfd, nc),
...                 ncf.ppf(0.99, dfn, dfd, nc), 100)
>>> ax.plot(x, ncf.pdf(x, dfn, dfd, nc),
...        'r-', lw=5, alpha=0.6, label='ncf pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = ncf(dfn, dfd, nc)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

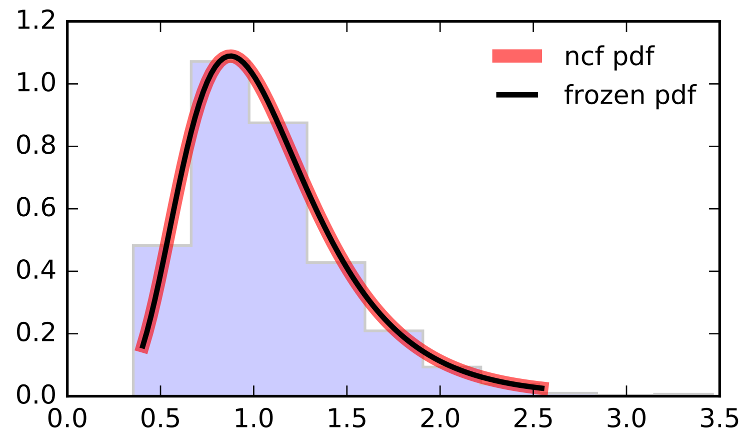
```
>>> vals = ncf.ppf([0.001, 0.5, 0.999], dfn, dfd, nc)
>>> np.allclose([0.001, 0.5, 0.999], ncf.cdf(vals, dfn, dfd, nc))
True
```

Generate random numbers:

```
>>> r = ncf.rvs(dfn, dfd, nc, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(dfn, dfd, nc, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, dfn, dfd, nc, loc=0, scale=1)`	Probability density function.
`logpdf(x, dfn, dfd, nc, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, dfn, dfd, nc, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, dfn, dfd, nc, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, dfn, dfd, nc, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, dfn, dfd, nc, loc=0, scale=1)`	Log of the survival function.
`ppf(q, dfn, dfd, nc, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, dfn, dfd, nc, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, dfn, dfd, nc, loc=0, scale=1)`	Non-central moment of order n
`stats(dfn, dfd, nc, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(dfn, dfd, nc, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, dfn, dfd, nc, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(dfn, dfd, nc), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(dfn, dfd, nc, loc=0, scale=1)`	Median of the distribution.
`mean(dfn, dfd, nc, loc=0, scale=1)`	Mean of the distribution.
`var(dfn, dfd, nc, loc=0, scale=1)`	Variance of the distribution.
`std(dfn, dfd, nc, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, dfn, dfd, nc, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**nct = <scipy.stats._continuous_distns.nct_gen object>**

A non-central Student's T continuous random variable.

As an instance of the *rv_continuous* class, *nct* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *nct* is:

```
                                    df**(df/2) * gamma(df+1)
nct.pdf(x, df, nc) = ----------------------------------------------------
                     2**df*exp(nc**2/2) * (df+x**2)**(df/2) * gamma(df/2)
```

for `df > 0`.

nct takes `df` and `nc` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `nct.pdf(x, df, nc, loc, scale)` is identically equivalent to `nct.pdf(y, df, nc) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import nct
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> df, nc = 14, 0.24
>>> mean, var, skew, kurt = nct.stats(df, nc, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(nct.ppf(0.01, df, nc),
...                 nct.ppf(0.99, df, nc), 100)
>>> ax.plot(x, nct.pdf(x, df, nc),
...        'r-', lw=5, alpha=0.6, label='nct pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = nct(df, nc)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

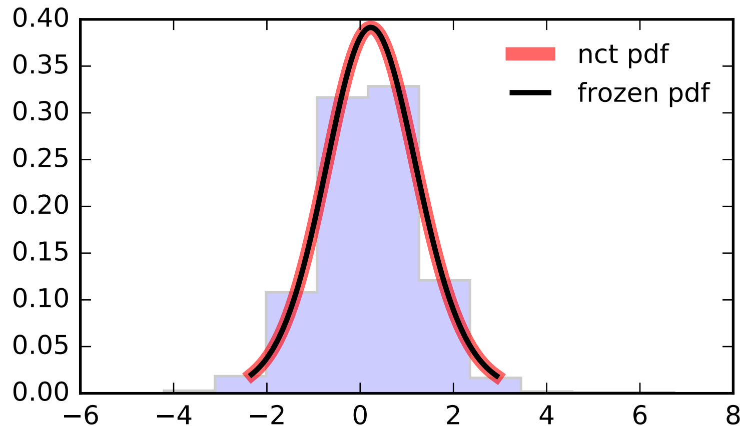
```
>>> vals = nct.ppf([0.001, 0.5, 0.999], df, nc)
>>> np.allclose([0.001, 0.5, 0.999], nct.cdf(vals, df, nc))
True
```

Generate random numbers:

```
>>> r = nct.rvs(df, nc, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(df, nc, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, df, nc, loc=0, scale=1)`	Probability density function.
`logpdf(x, df, nc, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, df, nc, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, df, nc, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, df, nc, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, df, nc, loc=0, scale=1)`	Log of the survival function.
`ppf(q, df, nc, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, df, nc, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, df, nc, loc=0, scale=1)`	Non-central moment of order n
`stats(df, nc, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(df, nc, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, df, nc, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(df, nc), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(df, nc, loc=0, scale=1)`	Median of the distribution.
`mean(df, nc, loc=0, scale=1)`	Mean of the distribution.
`var(df, nc, loc=0, scale=1)`	Variance of the distribution.
`std(df, nc, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, df, nc, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`norm`** **= <scipy.stats._continuous_distns.norm_gen object>**

A normal continuous random variable.

The location (loc) keyword specifies the mean. The scale (scale) keyword specifies the standard deviation.

As an instance of the *rv_continuous* class, *norm* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *norm* is:

```
norm.pdf(x) = exp(-x**2/2)/sqrt(2*pi)
```

The survival function, `norm.sf`, is also referred to as the Q-function in some contexts (see, e.g., Wikipedia's definition).

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `norm.pdf(x, loc, scale)` is identically equivalent to `norm.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import norm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = norm.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(norm.ppf(0.01),
...                 norm.ppf(0.99), 100)
>>> ax.plot(x, norm.pdf(x),
...        'r-', lw=5, alpha=0.6, label='norm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = norm()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

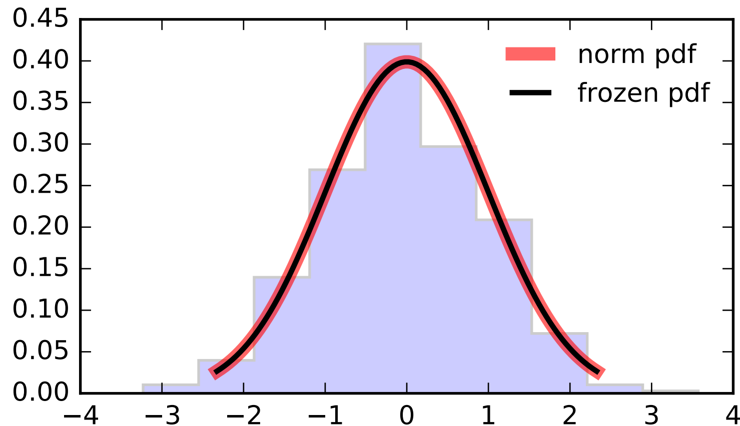
```
>>> vals = norm.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], norm.cdf(vals))
True
```

Generate random numbers:

```
>>> r = norm.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`pareto`** **= <scipy.stats._continuous_distns.pareto_gen object>**

A Pareto continuous random variable.

As an instance of the *`rv_continuous`* class, *`pareto`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `pareto` is:

```
pareto.pdf(x, b) = b / x**(b+1)
```

for `x >= 1`, `b > 0`.

`pareto` takes `b` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `pareto.pdf(x, b, loc, scale)` is identically equivalent to `pareto.pdf(y, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import pareto
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> b = 2.62
>>> mean, var, skew, kurt = pareto.stats(b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(pareto.ppf(0.01, b),
...                 pareto.ppf(0.99, b), 100)
>>> ax.plot(x, pareto.pdf(x, b),
...        'r-', lw=5, alpha=0.6, label='pareto pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = pareto(b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

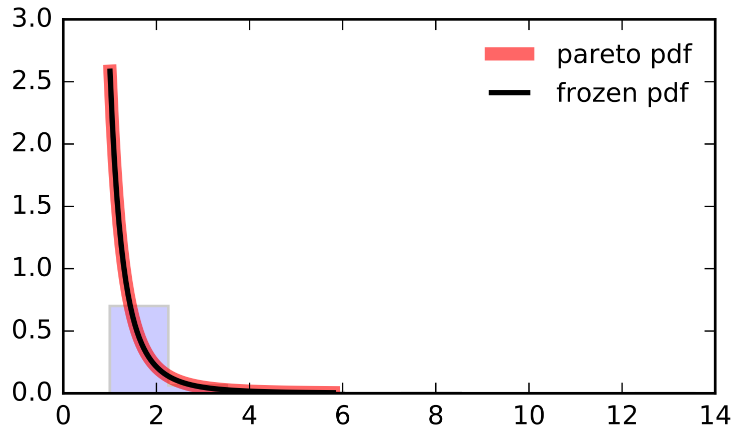
```
>>> vals = pareto.ppf([0.001, 0.5, 0.999], b)
>>> np.allclose([0.001, 0.5, 0.999], pareto.cdf(vals, b))
True
```

Generate random numbers:

```
>>> r = pareto.rvs(b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, b, loc=0, scale=1)`	Non-central moment of order n
`stats(b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(b,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(b, loc=0, scale=1)`	Median of the distribution.
`mean(b, loc=0, scale=1)`	Mean of the distribution.
`var(b, loc=0, scale=1)`	Variance of the distribution.
`std(b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**pearson3 = <scipy.stats._continuous_distns.pearson3_gen object>**

A pearson type III continuous random variable.

As an instance of the *rv_continuous* class, *pearson3* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *pearson3* is:

```
pearson3.pdf(x, skew) = abs(beta) / gamma(alpha) *
    (beta * (x - zeta))**(alpha - 1) * exp(-beta*(x - zeta))
```

where:

```
beta = 2 / (skew * stddev)
alpha = (stddev * beta)**2
zeta = loc - alpha / beta
```

pearson3 takes `skew` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `pearson3.pdf(x, skew, loc, scale)` is identically equivalent to `pearson3.pdf(y, skew) / scale` with `y = (x - loc) / scale`.

References

R.W. Vogel and D.E. McMartin, "Probability Plot Goodness-of-Fit and Skewness Estimation Procedures for the Pearson Type 3 Distribution", Water Resources Research, Vol.27, 3149-3158 (1991).

L.R. Salvosa, "Tables of Pearson's Type III Function", Ann. Math. Statist., Vol.1, 191-198 (1930).

"Using Modern Computing Tools to Fit the Pearson Type III Distribution to Aviation Loads Data", Office of Aviation Research (2003).

Examples

```
>>> from scipy.stats import pearson3
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> skew = 0.1
>>> mean, var, skew, kurt = pearson3.stats(skew, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(pearson3.ppf(0.01, skew),
...                 pearson3.ppf(0.99, skew), 100)
>>> ax.plot(x, pearson3.pdf(x, skew),
...        'r-', lw=5, alpha=0.6, label='pearson3 pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = pearson3(skew)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

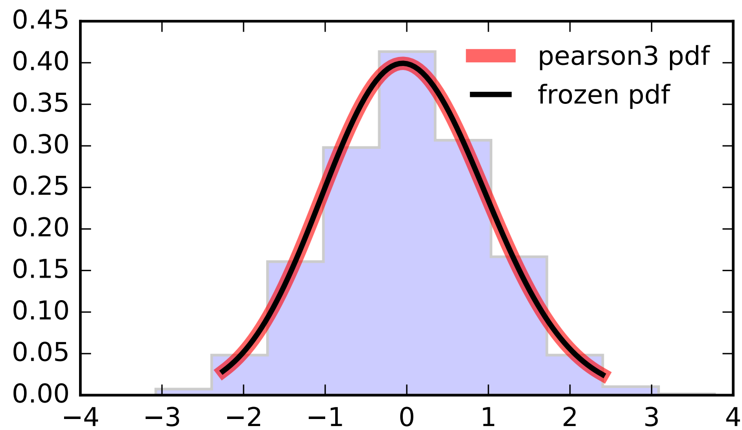
```
>>> vals = pearson3.ppf([0.001, 0.5, 0.999], skew)
>>> np.allclose([0.001, 0.5, 0.999], pearson3.cdf(vals, skew))
True
```

Generate random numbers:

```
>>> r = pearson3.rvs(skew, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(skew, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, skew, loc=0, scale=1)`	Probability density function.
`logpdf(x, skew, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, skew, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, skew, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, skew, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, skew, loc=0, scale=1)`	Log of the survival function.
`ppf(q, skew, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, skew, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, skew, loc=0, scale=1)`	Non-central moment of order n
`stats(skew, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(skew, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, skew, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(skew,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(skew, loc=0, scale=1)`	Median of the distribution.
`mean(skew, loc=0, scale=1)`	Mean of the distribution.
`var(skew, loc=0, scale=1)`	Variance of the distribution.
`std(skew, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, skew, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`powerlaw`** **= <scipy.stats._continuous_distns.powerlaw_gen object>**

A power-function continuous random variable.

As an instance of the *rv_continuous* class, *powerlaw* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *powerlaw* is:

```
powerlaw.pdf(x, a) = a * x**(a-1)
```

for `0 <= x <= 1`, `a > 0`.

powerlaw takes `a` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `powerlaw.pdf(x, a, loc, scale)` is identically equivalent to `powerlaw.pdf(y, a) / scale` with `y = (x - loc) / scale`.

powerlaw is a special case of *beta* with `b == 1`.

Examples

```
>>> from scipy.stats import powerlaw
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 1.66
>>> mean, var, skew, kurt = powerlaw.stats(a, moments='mvsk')
```

Display the probability density function (pdf):

```
>>> x = np.linspace(powerlaw.ppf(0.01, a),
...                 powerlaw.ppf(0.99, a), 100)
>>> ax.plot(x, powerlaw.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='powerlaw pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen pdf:

```
>>> rv = powerlaw(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of cdf and ppf:

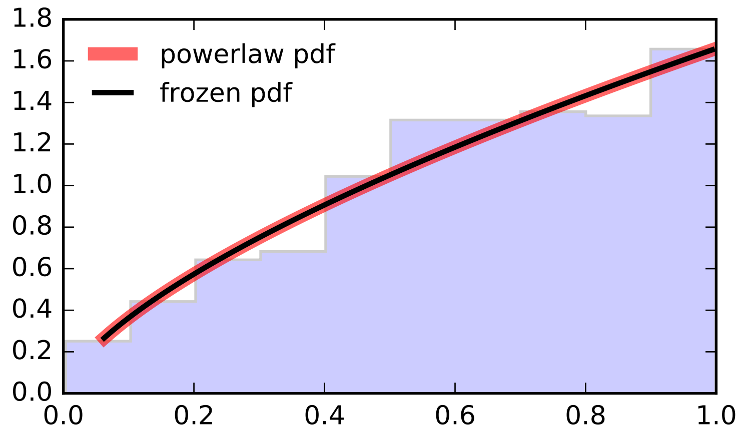
```
>>> vals = powerlaw.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], powerlaw.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = powerlaw.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`powerlognorm`** = **<scipy.stats._continuous_distns.powerlognorm_gen object>**

A power log-normal continuous random variable.

As an instance of the *`rv_continuous`* class, *`powerlognorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *powerlognorm* is:

```
powerlognorm.pdf(x, c, s) = c / (x*s) * phi(log(x)/s) *
                            (Phi(-log(x)/s))**(c-1),
```

where `phi` is the normal pdf, and `Phi` is the normal cdf, and `x > 0`, `s, c > 0`.

powerlognorm takes `c` and `s` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `powerlognorm.pdf(x, c, s, loc, scale)` is identically equivalent to `powerlognorm.pdf(y, c, s) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import powerlognorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c, s = 2.14, 0.446
>>> mean, var, skew, kurt = powerlognorm.stats(c, s, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(powerlognorm.ppf(0.01, c, s),
...                 powerlognorm.ppf(0.99, c, s), 100)
>>> ax.plot(x, powerlognorm.pdf(x, c, s),
...        'r-', lw=5, alpha=0.6, label='powerlognorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = powerlognorm(c, s)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

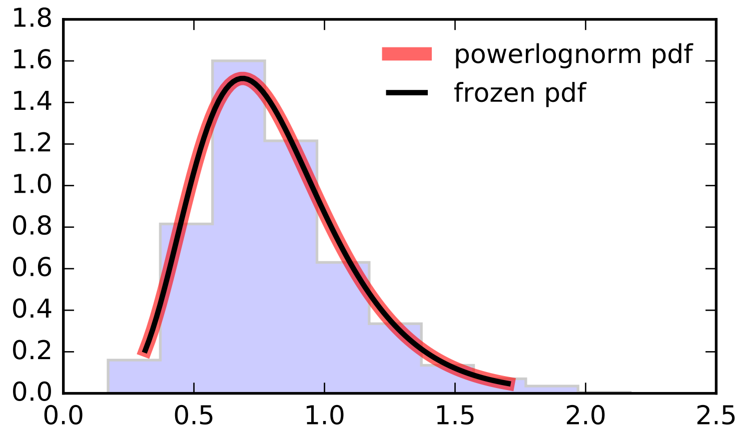
```
>>> vals = powerlognorm.ppf([0.001, 0.5, 0.999], c, s)
>>> np.allclose([0.001, 0.5, 0.999], powerlognorm.cdf(vals, c, s))
True
```

Generate random numbers:

```
>>> r = powerlognorm.rvs(c, s, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, s, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, s, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, s, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, s, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, s, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, s, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, s, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, s, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, s, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, s, loc=0, scale=1)`	Non-central moment of order n
`stats(c, s, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, s, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, s, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c, s), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, s, loc=0, scale=1)`	Median of the distribution.
`mean(c, s, loc=0, scale=1)`	Mean of the distribution.
`var(c, s, loc=0, scale=1)`	Variance of the distribution.
`std(c, s, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, s, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`powernorm`** **= <scipy.stats._continuous_distns.powernorm_gen object>**

A power normal continuous random variable.

As an instance of the *rv_continuous* class, *powernorm* object inherits from it a collection of generic

methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *powernorm* is:

```
powernorm.pdf(x, c) = c * phi(x) * (Phi(-x))**(c-1)
```

where `phi` is the normal pdf, and `Phi` is the normal cdf, and `x > 0`, `c > 0`.

powernorm takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `powernorm.pdf(x, c, loc, scale)` is identically equivalent to `powernorm.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import powernorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 4.45
>>> mean, var, skew, kurt = powernorm.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(powernorm.ppf(0.01, c),
...                 powernorm.ppf(0.99, c), 100)
>>> ax.plot(x, powernorm.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='powernorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = powernorm(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

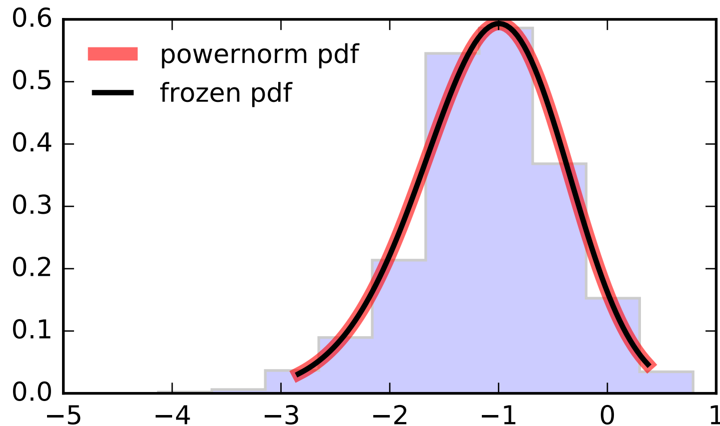
```
>>> vals = powernorm.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], powernorm.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = powernorm.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`rdist`** **= <scipy.stats._continuous_distns.rdist_gen object>**

An R-distributed continuous random variable.

As an instance of the *`rv_continuous`* class, *`rdist`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `rdist` is:

```
rdist.pdf(x, c) = (1-x**2)**(c/2-1) / B(1/2, c/2)
```

for `-1 <= x <= 1`, `c > 0`.

`rdist` takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `rdist.pdf(x, c, loc, scale)` is identically equivalent to `rdist.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import rdist
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.9
>>> mean, var, skew, kurt = rdist.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(rdist.ppf(0.01, c),
...                 rdist.ppf(0.99, c), 100)
>>> ax.plot(x, rdist.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='rdist pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = rdist(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

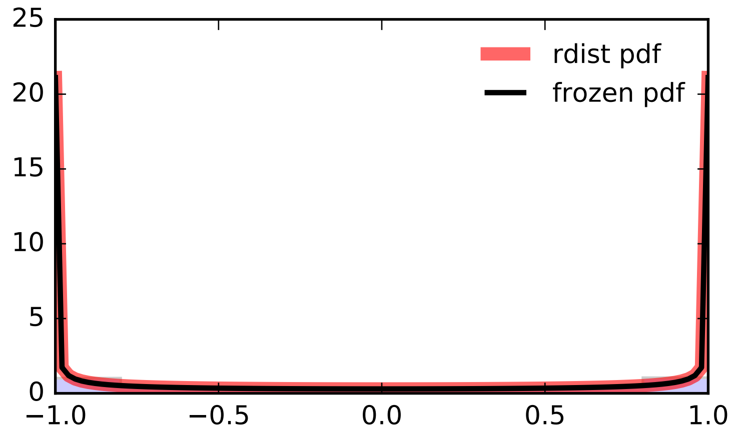
```
>>> vals = rdist.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], rdist.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = rdist.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`reciprocal`** **= <scipy.stats._continuous_distns.reciprocal_gen object>**

A reciprocal continuous random variable.

As an instance of the *`rv_continuous`* class, *`reciprocal`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `reciprocal` is:

```
reciprocal.pdf(x, a, b) = 1 / (x*log(b/a))
```

for `a <= x <= b, a, b > 0`.

`reciprocal` takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `reciprocal.pdf(x, a, b, loc, scale)` is identically equivalent to `reciprocal.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import reciprocal
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 0.00623, 1.01
>>> mean, var, skew, kurt = reciprocal.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(reciprocal.ppf(0.01, a, b),
...                 reciprocal.ppf(0.99, a, b), 100)
>>> ax.plot(x, reciprocal.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='reciprocal pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = reciprocal(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

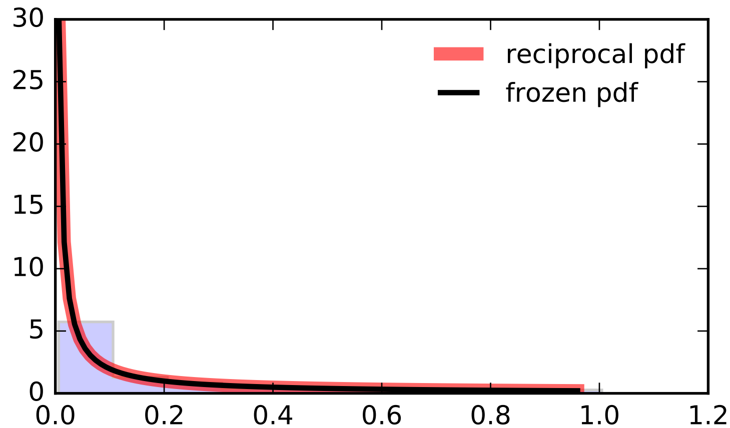
```
>>> vals = reciprocal.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], reciprocal.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = reciprocal.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, loc=0, scale=1, moments='mv')`	Mean(‘m’), variance(‘v’), skew(‘s’), and/or kurtosis(‘k’).
`entropy(a, b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`rayleigh`** **= <scipy.stats._continuous_distns.rayleigh_gen object>**

A Rayleigh continuous random variable.

As an instance of the *`rv_continuous`* class, *`rayleigh`* object inherits from it a collection of generic meth-

ods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `rayleigh` is:

```
rayleigh.pdf(r) = r * exp(-r**2/2)
```

for `x >= 0`.

`rayleigh` is a special case of `chi` with `df == 2`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `rayleigh.pdf(x, loc, scale)` is identically equivalent to `rayleigh.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import rayleigh
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = rayleigh.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(rayleigh.ppf(0.01),
...                 rayleigh.ppf(0.99), 100)
>>> ax.plot(x, rayleigh.pdf(x),
...        'r-', lw=5, alpha=0.6, label='rayleigh pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = rayleigh()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

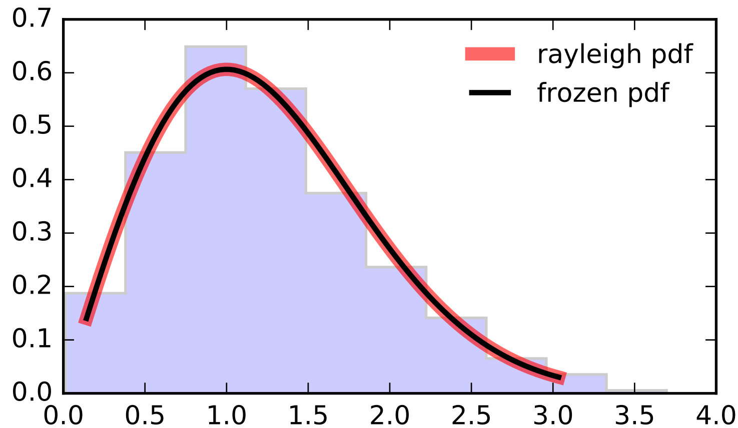
```
>>> vals = rayleigh.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], rayleigh.cdf(vals))
True
```

Generate random numbers:

```
>>> r = rayleigh.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`rice`** **= <scipy.stats._continuous_distns.rice_gen object>**

A Rice continuous random variable.

As an instance of the *`rv_continuous`* class, *`rice`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `rice` is:

```
rice.pdf(x, b) = x * exp(-(x**2+b**2)/2) * I[0](x*b)
```

for `x > 0`, `b > 0`.

`rice` takes `b` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `rice.pdf(x, b, loc, scale)` is identically equivalent to `rice.pdf(y, b) / scale` with `y = (x - loc) / scale`.

The Rice distribution describes the length, `r`, of a 2-D vector with components `(U+u, V+v)`, where `U, V` are constant, `u, v` are independent Gaussian random variables with standard deviation `s`. Let `R = (U**2 + V**2)**0.5`. Then the pdf of `r` is `rice.pdf(x, R/s, scale=s)`.

Examples

```
>>> from scipy.stats import rice
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> b = 0.775
>>> mean, var, skew, kurt = rice.stats(b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(rice.ppf(0.01, b),
...                 rice.ppf(0.99, b), 100)
>>> ax.plot(x, rice.pdf(x, b),
...        'r-', lw=5, alpha=0.6, label='rice pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = rice(b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

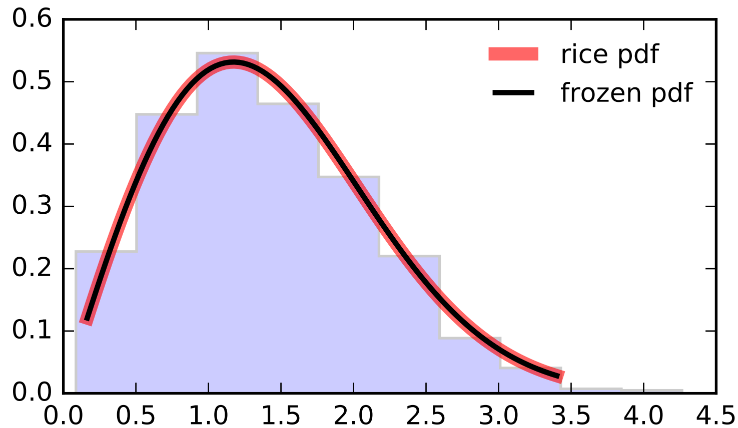
```
>>> vals = rice.ppf([0.001, 0.5, 0.999], b)
>>> np.allclose([0.001, 0.5, 0.999], rice.cdf(vals, b))
True
```

Generate random numbers:

```
>>> r = rice.rvs(b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, b, loc=0, scale=1)`	Non-central moment of order n
`stats(b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(b,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(b, loc=0, scale=1)`	Median of the distribution.
`mean(b, loc=0, scale=1)`	Mean of the distribution.
`var(b, loc=0, scale=1)`	Variance of the distribution.
`std(b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`recipinvgauss`** **= <scipy.stats._continuous_distns.recipinvgauss_gen object>**

A reciprocal inverse Gaussian continuous random variable.

As an instance of the *`rv_continuous`* class, *`recipinvgauss`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `recipinvgauss` is:

```
recipinvgauss.pdf(x, mu) = 1/sqrt(2*pi*x) * exp(-(1-mu*x)**2/(2*x*mu**2))
```

for `x >= 0`.

`recipinvgauss` takes `mu` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `recipinvgauss.pdf(x, mu, loc, scale)` is identically equivalent to `recipinvgauss.pdf(y, mu) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import recipinvgauss
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mu = 0.63
>>> mean, var, skew, kurt = recipinvgauss.stats(mu, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(recipinvgauss.ppf(0.01, mu),
...                 recipinvgauss.ppf(0.99, mu), 100)
>>> ax.plot(x, recipinvgauss.pdf(x, mu),
...        'r-', lw=5, alpha=0.6, label='recipinvgauss pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = recipinvgauss(mu)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

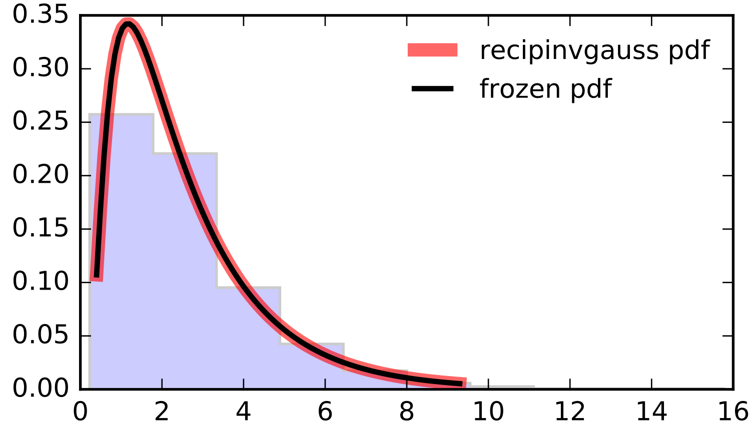
```
>>> vals = recipinvgauss.ppf([0.001, 0.5, 0.999], mu)
>>> np.allclose([0.001, 0.5, 0.999], recipinvgauss.cdf(vals, mu))
True
```

Generate random numbers:

```
>>> r = recipinvgauss.rvs(mu, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(mu, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, mu, loc=0, scale=1)`	Probability density function.
`logpdf(x, mu, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, mu, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, mu, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, mu, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, mu, loc=0, scale=1)`	Log of the survival function.
`ppf(q, mu, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, mu, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, mu, loc=0, scale=1)`	Non-central moment of order n
`stats(mu, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(mu, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, mu, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(mu,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(mu, loc=0, scale=1)`	Median of the distribution.
`mean(mu, loc=0, scale=1)`	Mean of the distribution.
`var(mu, loc=0, scale=1)`	Variance of the distribution.
`std(mu, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, mu, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`semicircular`** = **<scipy.stats._continuous_distns.semicircular_gen object>**

A semicircular continuous random variable.

As an instance of the *`rv_continuous`* class, *`semicircular`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *semicircular* is:

```
semicircular.pdf(x) = 2/pi * sqrt(1-x**2)
```

for `-1 <= x <= 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `semicircular.pdf(x, loc, scale)` is identically equivalent to `semicircular.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import semicircular
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = semicircular.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(semicircular.ppf(0.01),
...                 semicircular.ppf(0.99), 100)
>>> ax.plot(x, semicircular.pdf(x),
...        'r-', lw=5, alpha=0.6, label='semicircular pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = semicircular()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

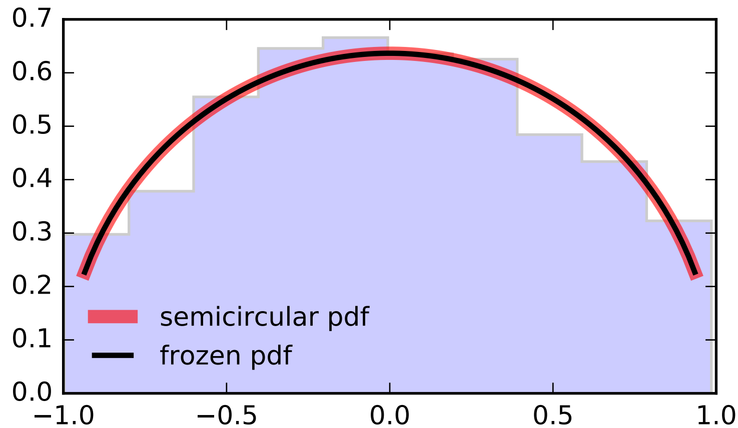
```
>>> vals = semicircular.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], semicircular.cdf(vals))
True
```

Generate random numbers:

```
>>> r = semicircular.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`skewnorm`** **= <scipy.stats._continuous_distns.skew_norm_gen object>**

A skew-normal random variable.

As an instance of the *`rv_continuous`* class, *`skewnorm`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The pdf is:

```
skewnorm.pdf(x, a) = 2*norm.pdf(x)*norm.cdf(ax)
```

`skewnorm` takes `a` as a skewness parameter When a=0 the distribution is identical to a normal distribution. rvs implements the method of *[R636]*.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `skewnorm.pdf(x, a, loc, scale)` is identically equivalent to `skewnorm.pdf(y, a) / scale` with `y = (x - loc) / scale`.

References

[R636]

Examples

```
>>> from scipy.stats import skewnorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 4
>>> mean, var, skew, kurt = skewnorm.stats(a, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(skewnorm.ppf(0.01, a),
...                 skewnorm.ppf(0.99, a), 100)
>>> ax.plot(x, skewnorm.pdf(x, a),
...        'r-', lw=5, alpha=0.6, label='skewnorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = skewnorm(a)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

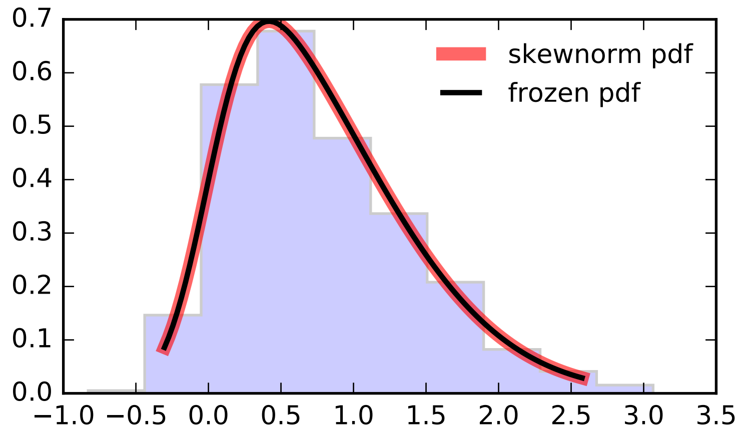
```
>>> vals = skewnorm.ppf([0.001, 0.5, 0.999], a)
>>> np.allclose([0.001, 0.5, 0.999], skewnorm.cdf(vals, a))
True
```

Generate random numbers:

```
>>> r = skewnorm.rvs(a, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, loc=0, scale=1)`	Non-central moment of order n
`stats(a, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0, scale=1)`	Median of the distribution.
`mean(a, loc=0, scale=1)`	Mean of the distribution.
`var(a, loc=0, scale=1)`	Variance of the distribution.
`std(a, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**t = <scipy.stats._continuous_distns.t_gen object>**

A Student's T continuous random variable.

As an instance of the `rv_continuous` class, `t` object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `t` is:

```
                                gamma((df+1)/2)
t.pdf(x, df) = ---------------------------------------------------
               sqrt(pi*df) * gamma(df/2) * (1+x**2/df)**((df+1)/2)
```

for `df > 0`.

`t` takes `df` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `t.pdf(x, df, loc, scale)` is identically equivalent to `t.pdf(y, df) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import t
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> df = 2.74
>>> mean, var, skew, kurt = t.stats(df, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(t.ppf(0.01, df),
...                 t.ppf(0.99, df), 100)
>>> ax.plot(x, t.pdf(x, df),
...        'r-', lw=5, alpha=0.6, label='t pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = t(df)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

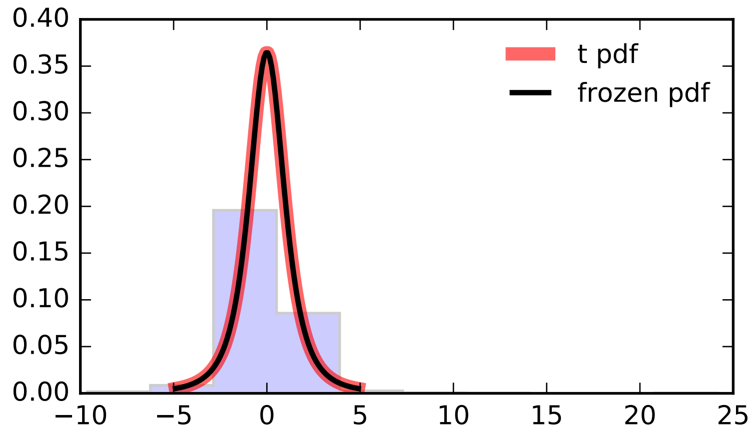
```
>>> vals = t.ppf([0.001, 0.5, 0.999], df)
>>> np.allclose([0.001, 0.5, 0.999], t.cdf(vals, df))
True
```

Generate random numbers:

```
>>> r = t.rvs(df, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(df, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, df, loc=0, scale=1)`	Probability density function.
`logpdf(x, df, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, df, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, df, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, df, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, df, loc=0, scale=1)`	Log of the survival function.
`ppf(q, df, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, df, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, df, loc=0, scale=1)`	Non-central moment of order n
`stats(df, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(df, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, df, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(df,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(df, loc=0, scale=1)`	Median of the distribution.
`mean(df, loc=0, scale=1)`	Mean of the distribution.
`var(df, loc=0, scale=1)`	Variance of the distribution.
`std(df, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, df, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`trapz`** **= <scipy.stats._continuous_distns.trapz_gen object>**

A trapezoidal continuous random variable.

As an instance of the *`rv_continuous`* class, *`trapz`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The trapezoidal distribution can be represented with an up-sloping line from `loc` to `(loc + c*scale)`, then constant to `(loc + d*scale)` and then downsloping from `(loc + d*scale)` to `(loc+scale)`.

trapz takes `c` and `d` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `trapz.pdf(x, c, d, loc, scale)` is identically equivalent to `trapz.pdf(y, c, d) / scale` with `y = (x - loc) / scale`.

The standard form is in the range [0, 1] with c the mode. The location parameter shifts the start to *loc*. The scale parameter changes the width from 1 to *scale*.

Examples

```
>>> from scipy.stats import trapz
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c, d = 0.2, 0.8
>>> mean, var, skew, kurt = trapz.stats(c, d, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(trapz.ppf(0.01, c, d),
...                 trapz.ppf(0.99, c, d), 100)
>>> ax.plot(x, trapz.pdf(x, c, d),
...        'r-', lw=5, alpha=0.6, label='trapz pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = trapz(c, d)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

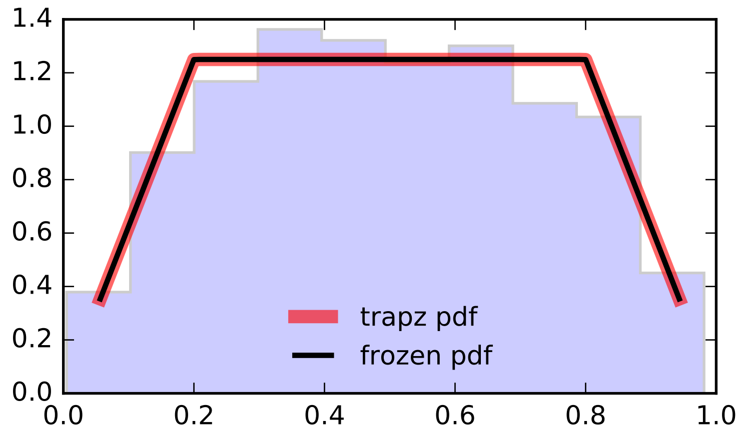
```
>>> vals = trapz.ppf([0.001, 0.5, 0.999], c, d)
>>> np.allclose([0.001, 0.5, 0.999], trapz.cdf(vals, c, d))
True
```

Generate random numbers:

```
>>> r = trapz.rvs(c, d, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, d, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, d, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, d, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, d, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, d, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, d, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, d, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, d, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, d, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, d, loc=0, scale=1)`	Non-central moment of order n
`stats(c, d, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, d, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, d, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c, d), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, d, loc=0, scale=1)`	Median of the distribution.
`mean(c, d, loc=0, scale=1)`	Mean of the distribution.
`var(c, d, loc=0, scale=1)`	Variance of the distribution.
`std(c, d, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, d, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`triang`** **= <scipy.stats._continuous_distns.triang_gen object>**

A triangular continuous random variable.

As an instance of the *`rv_continuous`* class, *`triang`* object inherits from it a collection of generic methods

(see below for the full list), and completes them with details specific for this particular distribution.

Notes

The triangular distribution can be represented with an up-sloping line from `loc` to `(loc + c*scale)` and then downsloping for `(loc + c*scale)` to `(loc+scale)`.

triang takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `triang.pdf(x, c, loc, scale)` is identically equivalent to `triang.pdf(y, c) / scale` with `y = (x - loc) / scale`.

The standard form is in the range [0, 1] with c the mode. The location parameter shifts the start to *loc*. The scale parameter changes the width from 1 to *scale*.

Examples

```
>>> from scipy.stats import triang
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.158
>>> mean, var, skew, kurt = triang.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(triang.ppf(0.01, c),
...                 triang.ppf(0.99, c), 100)
>>> ax.plot(x, triang.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='triang pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = triang(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

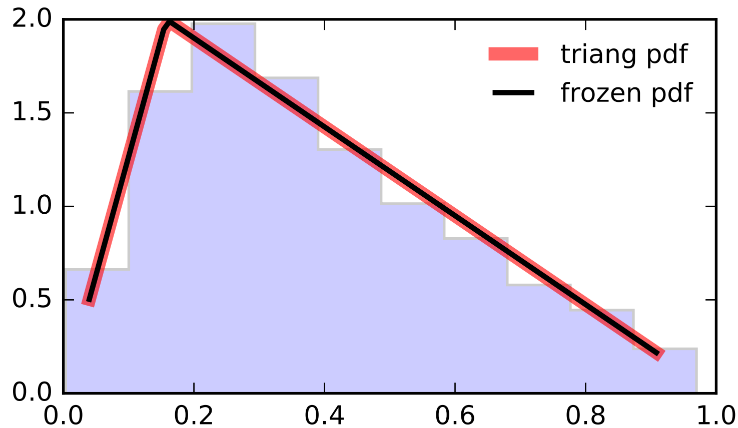
```
>>> vals = triang.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], triang.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = triang.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`truncexpon`** **= <scipy.stats._continuous_distns.truncexpon_gen object>**

A truncated exponential continuous random variable.

As an instance of the *`rv_continuous`* class, *`truncexpon`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *truncexpon* is:

```
truncexpon.pdf(x, b) = exp(-x) / (1-exp(-b))
```

for `0 < x < b`.

truncexpon takes `b` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `truncexpon.pdf(x, b, loc, scale)` is identically equivalent to `truncexpon.pdf(y, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import truncexpon
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> b = 4.69
>>> mean, var, skew, kurt = truncexpon.stats(b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(truncexpon.ppf(0.01, b),
...                 truncexpon.ppf(0.99, b), 100)
>>> ax.plot(x, truncexpon.pdf(x, b),
...        'r-', lw=5, alpha=0.6, label='truncexpon pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = truncexpon(b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

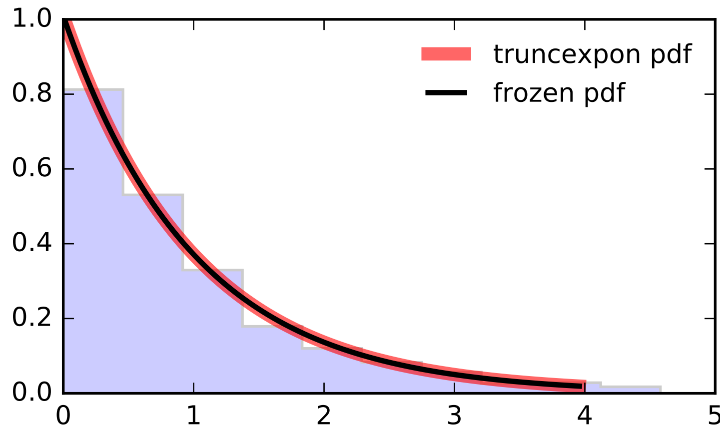
```
>>> vals = truncexpon.ppf([0.001, 0.5, 0.999], b)
>>> np.allclose([0.001, 0.5, 0.999], truncexpon.cdf(vals, b))
True
```

Generate random numbers:

```
>>> r = truncexpon.rvs(b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, b, loc=0, scale=1)`	Non-central moment of order n
`stats(b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(b,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(b, loc=0, scale=1)`	Median of the distribution.
`mean(b, loc=0, scale=1)`	Mean of the distribution.
`var(b, loc=0, scale=1)`	Variance of the distribution.
`std(b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`truncnorm`** **= <scipy.stats._continuous_distns.truncnorm_gen object>**

A truncated normal continuous random variable.

As an instance of the *rv_continuous* class, *truncnorm* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The standard form of this distribution is a standard normal truncated to the range [a, b] — notice that a and b are defined over the domain of the standard normal. To convert clip values for a specific mean and standard deviation, use:

```
a, b = (myclip_a - my_mean) / my_std, (myclip_b - my_mean) / my_std
```

`truncnorm` takes `a` and `b` as shape parameters.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `truncnorm.pdf(x, a, b, loc, scale)` is identically equivalent to `truncnorm.pdf(y, a, b) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import truncnorm
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a, b = 0.1, 2
>>> mean, var, skew, kurt = truncnorm.stats(a, b, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(truncnorm.ppf(0.01, a, b),
...                 truncnorm.ppf(0.99, a, b), 100)
>>> ax.plot(x, truncnorm.pdf(x, a, b),
...        'r-', lw=5, alpha=0.6, label='truncnorm pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = truncnorm(a, b)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

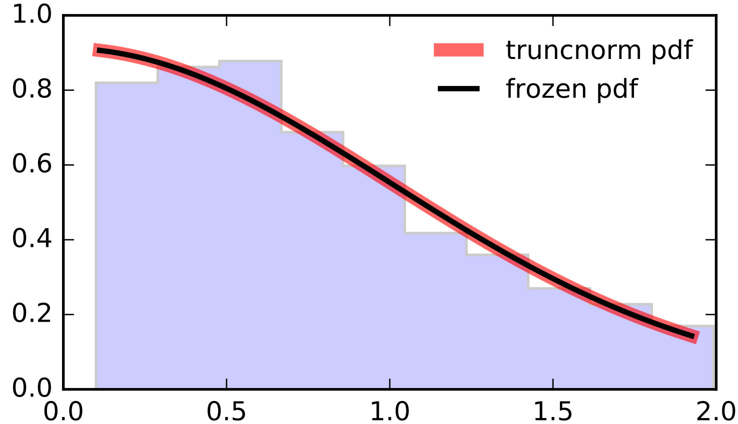
```
>>> vals = truncnorm.ppf([0.001, 0.5, 0.999], a, b)
>>> np.allclose([0.001, 0.5, 0.999], truncnorm.cdf(vals, a, b))
True
```

Generate random numbers:

```
>>> r = truncnorm.rvs(a, b, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(a, b, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, a, b, loc=0, scale=1)`	Probability density function.
`logpdf(x, a, b, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, a, b, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, a, b, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, a, b, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, a, b, loc=0, scale=1)`	Log of the survival function.
`ppf(q, a, b, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, b, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, a, b, loc=0, scale=1)`	Non-central moment of order n
`stats(a, b, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, b, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, a, b, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(a, b), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, b, loc=0, scale=1)`	Median of the distribution.
`mean(a, b, loc=0, scale=1)`	Mean of the distribution.
`var(a, b, loc=0, scale=1)`	Variance of the distribution.
`std(a, b, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, a, b, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`tukeylambda`** **= <scipy.stats._continuous_distns.tukeylambda_gen object>**

A Tukey-Lamdba continuous random variable.

As an instance of the *`rv_continuous`* class, *`tukeylambda`* object inherits from it a collection of generic

methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

A flexible distribution, able to represent and interpolate between the following distributions:

- Cauchy (lam=-1)
- logistic (lam=0.0)
- approx Normal (lam=0.14)
- u-shape (lam = 0.5)
- uniform from -1 to 1 (lam = 1)

`tukeylambda` takes `lam` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `tukeylambda.pdf(x, lam, loc, scale)` is identically equivalent to `tukeylambda.pdf(y, lam) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import tukeylambda
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> lam = 3.13
>>> mean, var, skew, kurt = tukeylambda.stats(lam, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(tukeylambda.ppf(0.01, lam),
...                 tukeylambda.ppf(0.99, lam), 100)
>>> ax.plot(x, tukeylambda.pdf(x, lam),
...        'r-', lw=5, alpha=0.6, label='tukeylambda pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = tukeylambda(lam)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

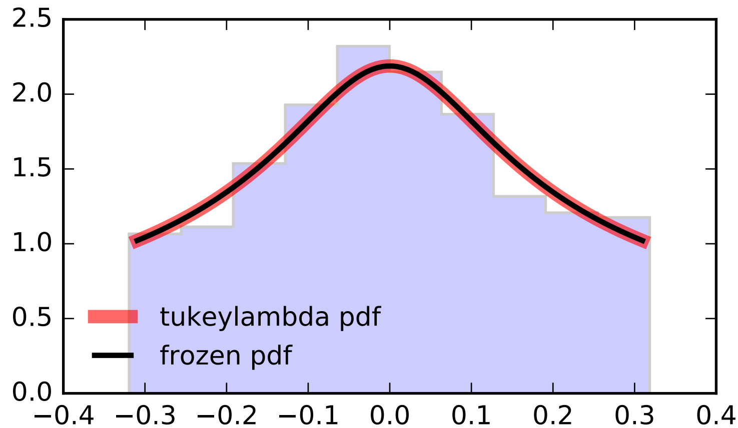
```
>>> vals = tukeylambda.ppf([0.001, 0.5, 0.999], lam)
>>> np.allclose([0.001, 0.5, 0.999], tukeylambda.cdf(vals, lam))
True
```

Generate random numbers:

```
>>> r = tukeylambda.rvs(lam, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(lam, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, lam, loc=0, scale=1)`	Probability density function.
`logpdf(x, lam, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, lam, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, lam, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, lam, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, lam, loc=0, scale=1)`	Log of the survival function.
`ppf(q, lam, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, lam, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, lam, loc=0, scale=1)`	Non-central moment of order n
`stats(lam, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(lam, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, lam, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(lam,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(lam, loc=0, scale=1)`	Median of the distribution.
`mean(lam, loc=0, scale=1)`	Mean of the distribution.
`var(lam, loc=0, scale=1)`	Variance of the distribution.
`std(lam, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, lam, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`uniform`** **= <scipy.stats._continuous_distns.uniform_gen object>**

A uniform continuous random variable.

This distribution is constant between *loc* and `loc + scale`.

As an instance of the `rv_continuous` class, `uniform` object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Examples

```
>>> from scipy.stats import uniform
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = uniform.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(uniform.ppf(0.01),
...                 uniform.ppf(0.99), 100)
>>> ax.plot(x, uniform.pdf(x),
...        'r-', lw=5, alpha=0.6, label='uniform pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = uniform()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

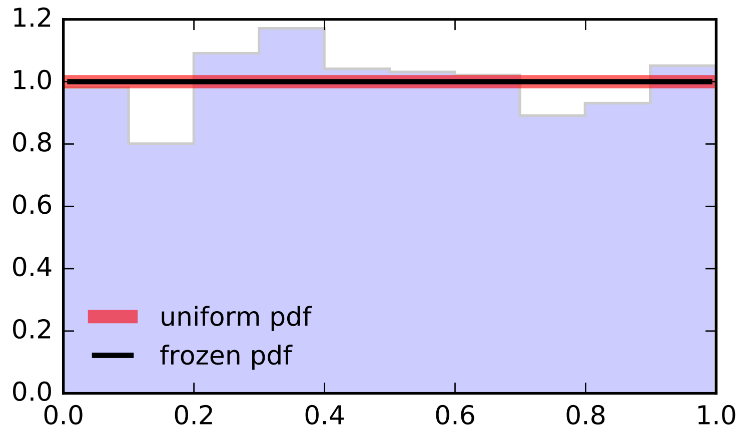
```
>>> vals = uniform.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], uniform.cdf(vals))
True
```

Generate random numbers:

```
>>> r = uniform.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`vonmises`** **= <scipy.stats._continuous_distns.vonmises_gen object>**

A Von Mises continuous random variable.

As an instance of the *`rv_continuous`* class, *`vonmises`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

vonmises_line
The same distribution, defined on a [-pi, pi] segment of the real line.

Notes

If *x* is not in range or *loc* is not in range it assumes they are angles and converts them to [-pi, pi] equivalents.

The probability density function for `vonmises` is:

```
vonmises.pdf(x, kappa) = exp(kappa * cos(x)) / (2*pi*I[0](kappa))
```

for `-pi <= x <= pi`, `kappa > 0`.

`vonmises` takes `kappa` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `vonmises.pdf(x, kappa, loc, scale)` is identically equivalent to `vonmises.pdf(y, kappa) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import vonmises
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> kappa = 3.99
>>> mean, var, skew, kurt = vonmises.stats(kappa, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(vonmises.ppf(0.01, kappa),
...                 vonmises.ppf(0.99, kappa), 100)
>>> ax.plot(x, vonmises.pdf(x, kappa),
...        'r-', lw=5, alpha=0.6, label='vonmises pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = vonmises(kappa)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

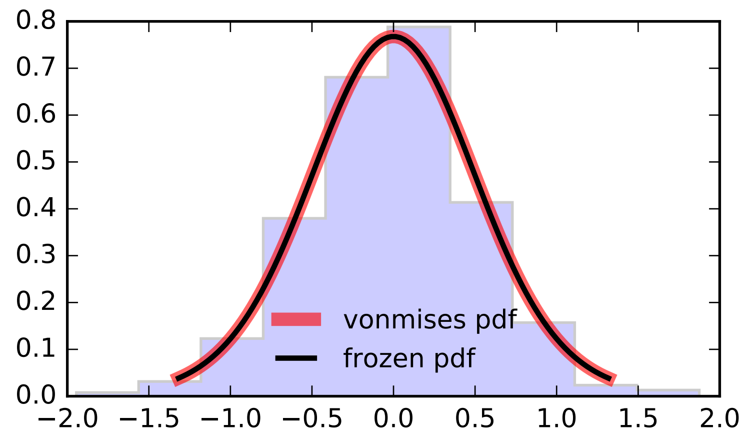
```
>>> vals = vonmises.ppf([0.001, 0.5, 0.999], kappa)
>>> np.allclose([0.001, 0.5, 0.999], vonmises.cdf(vals, kappa))
True
```

Generate random numbers:

```
>>> r = vonmises.rvs(kappa, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(kappa, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, kappa, loc=0, scale=1)`	Probability density function.
`logpdf(x, kappa, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, kappa, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, kappa, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, kappa, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, kappa, loc=0, scale=1)`	Log of the survival function.
`ppf(q, kappa, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, kappa, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, kappa, loc=0, scale=1)`	Non-central moment of order n
`stats(kappa, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(kappa, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, kappa, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(kappa,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(kappa, loc=0, scale=1)`	Median of the distribution.
`mean(kappa, loc=0, scale=1)`	Mean of the distribution.
`var(kappa, loc=0, scale=1)`	Variance of the distribution.
`std(kappa, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, kappa, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`vonmises_line`** **= <scipy.stats._continuous_distns.vonmises_gen object>**

A Von Mises continuous random variable.

As an instance of the *rv_continuous* class, *vonmises_line* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

vonmises_line
The same distribution, defined on a [-pi, pi] segment of the real line.

Notes

If *x* is not in range or *loc* is not in range it assumes they are angles and converts them to [-pi, pi] equivalents.

The probability density function for *vonmises* is:

```
vonmises.pdf(x, kappa) = exp(kappa * cos(x)) / (2*pi*I[0](kappa))
```

for `-pi <= x <= pi`, `kappa > 0`.

vonmises takes `kappa` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `vonmises_line.pdf(x, kappa, loc, scale)` is identically equivalent to `vonmises_line.pdf(y, kappa) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import vonmises_line
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> kappa = 3.99
>>> mean, var, skew, kurt = vonmises_line.stats(kappa, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(vonmises_line.ppf(0.01, kappa),
...                 vonmises_line.ppf(0.99, kappa), 100)
>>> ax.plot(x, vonmises_line.pdf(x, kappa),
...        'r-', lw=5, alpha=0.6, label='vonmises_line pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = vonmises_line(kappa)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

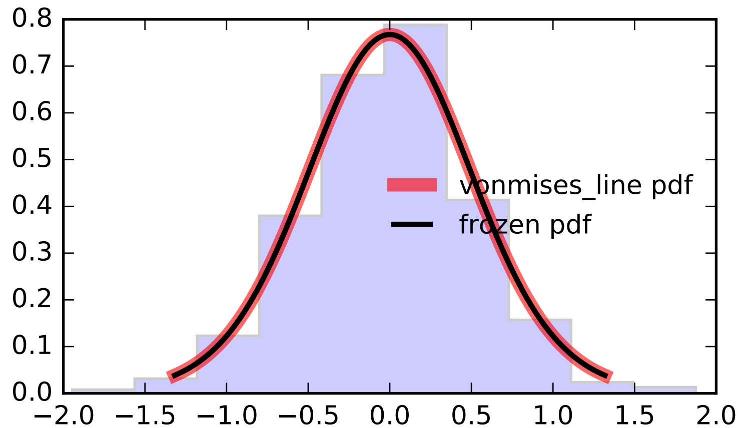
```
>>> vals = vonmises_line.ppf([0.001, 0.5, 0.999], kappa)
>>> np.allclose([0.001, 0.5, 0.999], vonmises_line.cdf(vals, kappa))
True
```

Generate random numbers:

```
>>> r = vonmises_line.rvs(kappa, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(kappa, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, kappa, loc=0, scale=1)`	Probability density function.
`logpdf(x, kappa, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, kappa, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, kappa, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, kappa, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, kappa, loc=0, scale=1)`	Log of the survival function.
`ppf(q, kappa, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, kappa, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, kappa, loc=0, scale=1)`	Non-central moment of order n
`stats(kappa, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(kappa, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, kappa, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(kappa,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(kappa, loc=0, scale=1)`	Median of the distribution.
`mean(kappa, loc=0, scale=1)`	Mean of the distribution.
`var(kappa, loc=0, scale=1)`	Variance of the distribution.
`std(kappa, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, kappa, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`wald`** **= <scipy.stats._continuous_distns.wald_gen object>**

A Wald continuous random variable.

As an instance of the *`rv_continuous`* class, *`wald`* object inherits from it a collection of generic methods

(see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for `wald` is:

```
wald.pdf(x) = 1/sqrt(2*pi*x**3) * exp(-(x-1)**2/(2*x))
```

for `x > 0`.

`wald` is a special case of `invgauss` with `mu == 1`.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `wald.pdf(x, loc, scale)` is identically equivalent to `wald.pdf(y) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import wald
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mean, var, skew, kurt = wald.stats(moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(wald.ppf(0.01),
...                 wald.ppf(0.99), 100)
>>> ax.plot(x, wald.pdf(x),
...        'r-', lw=5, alpha=0.6, label='wald pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = wald()
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

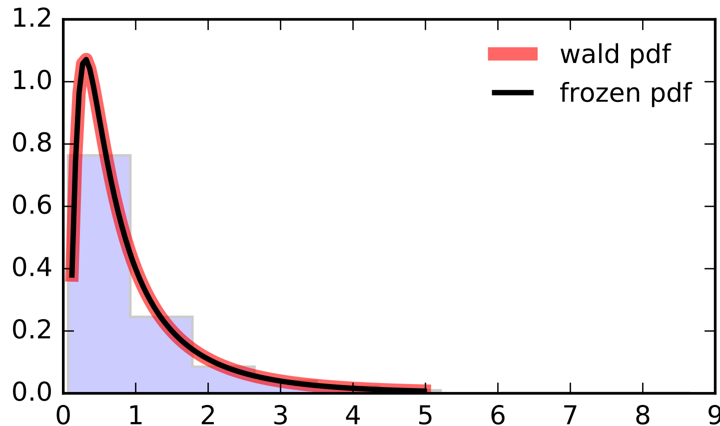
```
>>> vals = wald.ppf([0.001, 0.5, 0.999])
>>> np.allclose([0.001, 0.5, 0.999], wald.cdf(vals))
True
```

Generate random numbers:

```
>>> r = wald.rvs(size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, loc=0, scale=1)`	Probability density function.
`logpdf(x, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, loc=0, scale=1)`	Log of the survival function.
`ppf(q, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, loc=0, scale=1)`	Non-central moment of order n
`stats(loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(loc=0, scale=1)`	Median of the distribution.
`mean(loc=0, scale=1)`	Mean of the distribution.
`var(loc=0, scale=1)`	Variance of the distribution.
`std(loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`weibull_min`** **= <scipy.stats._continuous_distns.frechet_r_gen object>**

A Frechet right (or Weibull minimum) continuous random variable.

As an instance of the *`rv_continuous`* class, *`weibull_min`* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

weibull_min
The same distribution as *frechet_r*.

frechet_l, *weibull_max*

Notes

The probability density function for *frechet_r* is:

```
frechet_r.pdf(x, c) = c * x**(c-1) * exp(-x**c)
```

for `x > 0`, `c > 0`.

frechet_r takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `weibull_min.pdf(x, c, loc, scale)` is identically equivalent to `weibull_min.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import weibull_min
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 1.79
>>> mean, var, skew, kurt = weibull_min.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(weibull_min.ppf(0.01, c),
...                 weibull_min.ppf(0.99, c), 100)
>>> ax.plot(x, weibull_min.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='weibull_min pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = weibull_min(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

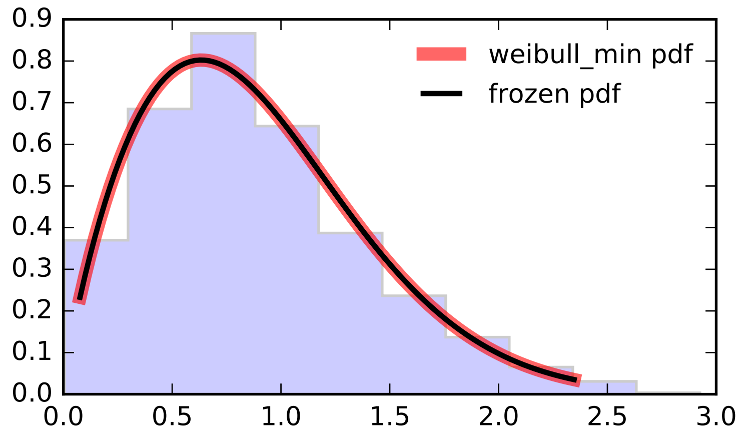
```
>>> vals = weibull_min.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], weibull_min.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = weibull_min.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`weibull_max`** = **<scipy.stats._continuous_distns.frechet_l_gen object>**

A Frechet left (or Weibull maximum) continuous random variable.

As an instance of the *rv_continuous* class, *weibull_max* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

See also:

weibull_max
The same distribution as *frechet_l*.

frechet_r, *weibull_min*

Notes

The probability density function for *frechet_l* is:

```
frechet_l.pdf(x, c) = c * (-x)**(c-1) * exp(-(-x)**c)
```

for `x < 0`, `c > 0`.

frechet_l takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `weibull_max.pdf(x, c, loc, scale)` is identically equivalent to `weibull_max.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import weibull_max
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 2.87
>>> mean, var, skew, kurt = weibull_max.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(weibull_max.ppf(0.01, c),
...                 weibull_max.ppf(0.99, c), 100)
>>> ax.plot(x, weibull_max.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='weibull_max pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = weibull_max(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

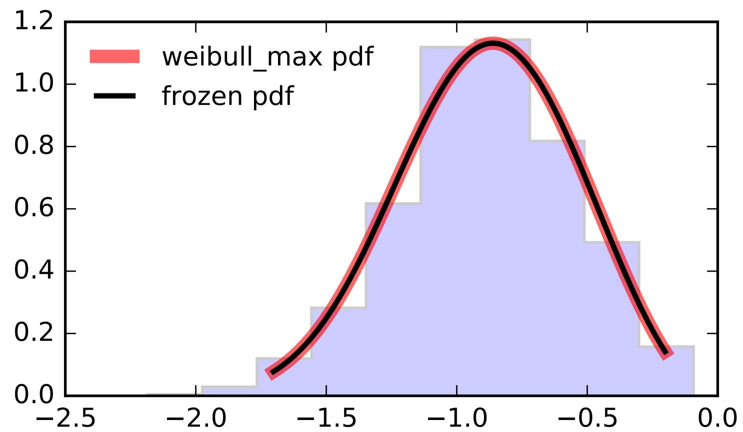
```
>>> vals = weibull_max.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], weibull_max.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = weibull_max.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

`rvs(c, loc=0, scale=1, size=1, random_state=None)`	Random variates.
`pdf(x, c, loc=0, scale=1)`	Probability density function.
`logpdf(x, c, loc=0, scale=1)`	Log of the probability density function.
`cdf(x, c, loc=0, scale=1)`	Cumulative distribution function.
`logcdf(x, c, loc=0, scale=1)`	Log of the cumulative distribution function.
`sf(x, c, loc=0, scale=1)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(x, c, loc=0, scale=1)`	Log of the survival function.
`ppf(q, c, loc=0, scale=1)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, c, loc=0, scale=1)`	Inverse survival function (inverse of `sf`).
`moment(n, c, loc=0, scale=1)`	Non-central moment of order n
`stats(c, loc=0, scale=1, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(c, loc=0, scale=1)`	(Differential) entropy of the RV.
`fit(data, c, loc=0, scale=1)`	Parameter estimates for generic data.
`expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)`	Expected value of a function (of one argument) with respect to the distribution.
`median(c, loc=0, scale=1)`	Median of the distribution.
`mean(c, loc=0, scale=1)`	Mean of the distribution.
`var(c, loc=0, scale=1)`	Variance of the distribution.
`std(c, loc=0, scale=1)`	Standard deviation of the distribution.
`interval(alpha, c, loc=0, scale=1)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**wrapcauchy = <scipy.stats._continuous_distns.wrapcauchy_gen object>**

A wrapped Cauchy continuous random variable.

As an instance of the *rv_continuous* class, *wrapcauchy* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability density function for *wrapcauchy* is:

```
wrapcauchy.pdf(x, c) = (1-c**2) / (2*pi*(1+c**2-2*c*cos(x)))
```

for `0 <= x <= 2*pi`, `0 < c < 1`.

wrapcauchy takes `c` as a shape parameter.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `wrapcauchy.pdf(x, c, loc, scale)` is identically equivalent to `wrapcauchy.pdf(y, c) / scale` with `y = (x - loc) / scale`.

Examples

```
>>> from scipy.stats import wrapcauchy
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> c = 0.0311
>>> mean, var, skew, kurt = wrapcauchy.stats(c, moments='mvsk')
```

Display the probability density function (`pdf`):

```
>>> x = np.linspace(wrapcauchy.ppf(0.01, c),
...                 wrapcauchy.ppf(0.99, c), 100)
>>> ax.plot(x, wrapcauchy.pdf(x, c),
...        'r-', lw=5, alpha=0.6, label='wrapcauchy pdf')
```

Alternatively, the distribution object can be called (as a function) to fix the shape, location and scale parameters. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pdf`:

```
>>> rv = wrapcauchy(c)
>>> ax.plot(x, rv.pdf(x), 'k-', lw=2, label='frozen pdf')
```

Check accuracy of `cdf` and `ppf`:

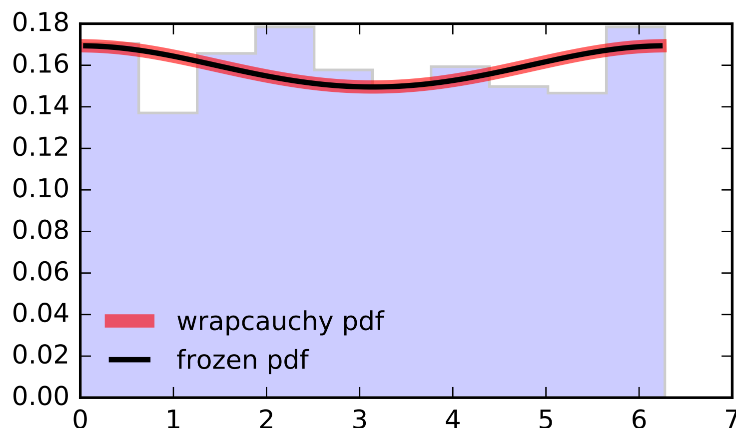
```
>>> vals = wrapcauchy.ppf([0.001, 0.5, 0.999], c)
>>> np.allclose([0.001, 0.5, 0.999], wrapcauchy.cdf(vals, c))
True
```

Generate random numbers:

```
>>> r = wrapcauchy.rvs(c, size=1000)
```

And compare the histogram:

```
>>> ax.hist(r, normed=True, histtype='stepfilled', alpha=0.2)
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Methods

rvs(c, loc=0, scale=1, size=1, random_state=None)	Random variates.
pdf(x, c, loc=0, scale=1)	Probability density function.
logpdf(x, c, loc=0, scale=1)	Log of the probability density function.
cdf(x, c, loc=0, scale=1)	Cumulative distribution function.
logcdf(x, c, loc=0, scale=1)	Log of the cumulative distribution function.
sf(x, c, loc=0, scale=1)	Survival function (also defined as 1 - cdf, but *sf* is sometimes more accurate).
logsf(x, c, loc=0, scale=1)	Log of the survival function.
ppf(q, c, loc=0, scale=1)	Percent point function (inverse of cdf — percentiles).
isf(q, c, loc=0, scale=1)	Inverse survival function (inverse of sf).
moment(n, c, loc=0, scale=1)	Non-central moment of order n
stats(c, loc=0, scale=1, moments='mv')	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
entropy(c, loc=0, scale=1)	(Differential) entropy of the RV.
fit(data, c, loc=0, scale=1)	Parameter estimates for generic data.
expect(func, args=(c,), loc=0, scale=1, lb=None, ub=None, conditional=False, **kwds)	Expected value of a function (of one argument) with respect to the distribution.
median(c, loc=0, scale=1)	Median of the distribution.
mean(c, loc=0, scale=1)	Mean of the distribution.
var(c, loc=0, scale=1)	Variance of the distribution.
std(c, loc=0, scale=1)	Standard deviation of the distribution.
interval(alpha, c, loc=0, scale=1)	Endpoints of the range that contains alpha percent of the distribution

5.27.2 Multivariate distributions

multivariate_normal	A multivariate normal random variable.
matrix_normal	A matrix normal random variable.
dirichlet	A Dirichlet random variable.
wishart	A Wishart random variable.
invwishart	An inverse Wishart random variable.
multinomial	A multinomial random variable.
special_ortho_group	A matrix-valued SO(N) random variable.
ortho_group	A matrix-valued O(N) random variable.
random_correlation	A random correlation matrix.

scipy.stats.**multivariate_normal = <scipy.stats._multivariate.multivariate_normal_gen object>**

A multivariate normal random variable.

The *mean* keyword specifies the mean. The *cov* keyword specifies the covariance matrix.

Parameters

x : array_like
: Quantiles, with the last axis of *x* denoting the components.

mean : array_like, optional
: Mean of the distribution (default zero)

cov : array_like, optional

Covariance matrix of the distribution (default one)

allow_singular : bool, optional

Whether to allow a singular covariance matrix. (Default: False)

random_state : None or int or np.random.RandomState instance, optional

If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

Alternatively, the object may be called (as a function) to fix the mean and covariance parameters, returning a "frozen" multivariate normal random variable:

rv = multivariate_normal(mean=None, cov=1, allow_singular=False)

- Frozen object with the same methods but holding the given mean and covariance fixed.

Notes

***Setting the parameter* mean *to* None *is equivalent to having* mean**

be the zero-vector. The parameter *cov* can be a scalar, in which case the covariance matrix is the identity times that value, a vector of diagonal entries for the covariance matrix, or a two-dimensional array_like.

The covariance matrix *cov* must be a (symmetric) positive semi-definite matrix. The determinant and inverse of *cov* are computed as the pseudo-determinant and pseudo-inverse, respectively, so that *cov* does not need to have full rank.

The probability density function for `multivariate_normal` is

$$f(x) = \frac{1}{\sqrt{(2\pi)^k \det \Sigma}} \exp\left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right),$$

where μ is the mean, Σ the covariance matrix, and k is the dimension of the space where x takes values.

New in version 0.14.0.

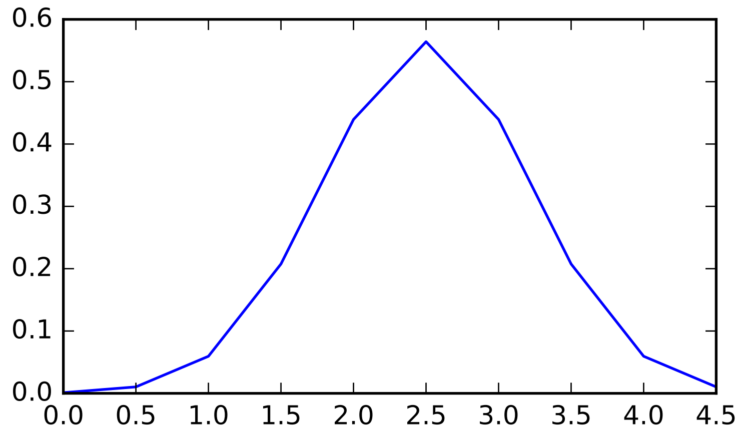
Examples

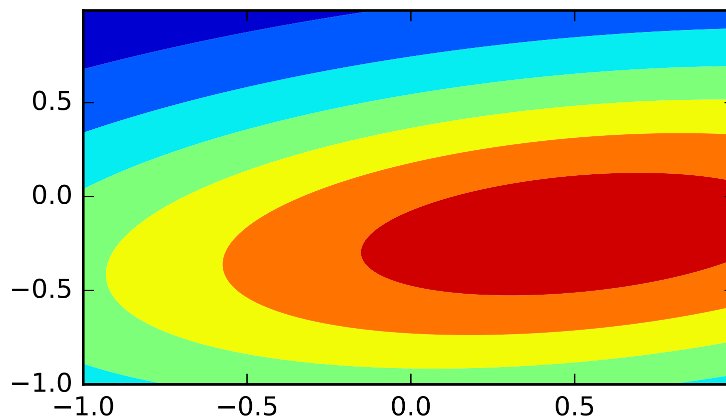
```
>>> import matplotlib.pyplot as plt
>>> from scipy.stats import multivariate_normal
```

```
>>> x = np.linspace(0, 5, 10, endpoint=False)
>>> y = multivariate_normal.pdf(x, mean=2.5, cov=0.5); y
array([ 0.00108914,  0.01033349,  0.05946514,  0.20755375,  0.43939129,
        0.56418958,  0.43939129,  0.20755375,  0.05946514,  0.01033349])
>>> fig1 = plt.figure()
>>> ax = fig1.add_subplot(111)
>>> ax.plot(x, y)
```

The input quantiles can be any shape of array, as long as the last axis labels the components. This allows us for instance to display the frozen pdf for a non-isotropic random variable in 2D as follows:

```
>>> x, y = np.mgrid[-1:1:.01, -1:1:.01]
>>> pos = np.dstack((x, y))
>>> rv = multivariate_normal([0.5, -0.2], [[2.0, 0.3], [0.3, 0.5]])
>>> fig2 = plt.figure()
>>> ax2 = fig2.add_subplot(111)
>>> ax2.contourf(x, y, rv.pdf(pos))
```





Methods

`pdf(x, mean=None, cov=1, allow_singular=False)`	Probability density function.
`logpdf(x, mean=None, cov=1, allow_singular=False)`	Log of the probability density function.
`rvs(mean=None, cov=1, size=1, random_state=None)`	Draw random samples from a multivariate normal distribution.
`entropy()`	Compute the differential entropy of the multivariate normal.

`scipy.stats.`**`matrix_normal`** **= <scipy.stats._multivariate.matrix_normal_gen object>**

A matrix normal random variable.

The *mean* keyword specifies the mean. The *rowcov* keyword specifies the among-row covariance matrix. The 'colcov' keyword specifies the among-column covariance matrix.

Parameters **X** : array_like

Quantiles, with the last two axes of *X* denoting the components.

mean : array_like, optional

Mean of the distribution (default: *None*)

rowcov : array_like, optional

Among-row covariance matrix of the distribution (default: *1*)

colcov : array_like, optional

Among-column covariance matrix of the distribution (default: *1*)

random_state : None or int or np.random.RandomState instance, optional

If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

Alternatively, the object may be called (as a function) to fix the mean and covariance parameters, returning a "frozen" matrix normal random variable:

rv = matrix_normal(mean=None, rowcov=1, colcov=1)

- Frozen object with the same methods but holding the given mean and covariance fixed.

Notes

If* mean *is set to* None *then a matrix of zeros is used for the mean.

The dimensions of this matrix are inferred from the shape of *rowcov* and *colcov*, if these are provided, or set to *1* if ambiguous.

rowcov and *colcov* can be two-dimensional array_likes specifying the covariance matrices directly. Alternatively, a one-dimensional array will be be interpreted as the entries of a diagonal matrix, and a scalar or zero-dimensional array will be interpreted as this value times the identity matrix.

The covariance matrices specified by *rowcov* and *colcov* must be (symmetric) positive definite. If the samples in *X* are $m \times n$, then *rowcov* must be $m \times m$ and *colcov* must be $n \times n$. *mean* must be the same shape as *X*.

The probability density function for `matrix_normal` is

$$f(X) = (2\pi)^{-\frac{mn}{2}}|U|^{-\frac{n}{2}}|V|^{-\frac{m}{2}} \exp\left(-\frac{1}{2}\mathrm{Tr}\left[U^{-1}(X-M)V^{-1}(X-M)^T\right]\right),$$

where M is the mean, U the among-row covariance matrix, V the among-column covariance matrix.

The *allow_singular* behaviour of the `multivariate_normal` distribution is not currently supported. Covariance matrices must be full rank.

The `matrix_normal` distribution is closely related to the `multivariate_normal` distribution. Specifically, $\mathrm{Vec}(X)$ (the vector formed by concatenating the columns of X) has a multivariate normal distribution with mean $\mathrm{Vec}(M)$ and covariance $V \otimes U$ (where $\otimes$ is the Kronecker product). Sampling and pdf evaluation are $\mathcal{O}(m^3 + n^3 + m^2 n + m n^2)$ for the matrix normal, but $\mathcal{O}(m^3 n^3)$ for the equivalent multivariate normal, making this equivalent form algorithmically inefficient.

New in version 0.17.0.

Examples

```
>>> from scipy.stats import matrix_normal
```

```
>>> M = np.arange(6).reshape(3,2); M
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> U = np.diag([1,2,3]); U
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>> V = 0.3*np.identity(2); V
array([[ 0.3,  0. ],
       [ 0. ,  0.3]])
>>> X = M + 0.1; X
array([[ 0.1,  1.1],
       [ 2.1,  3.1],
       [ 4.1,  5.1]])
>>> matrix_normal.pdf(X, mean=M, rowcov=U, colcov=V)
0.023410202050005054
```

```
>>> # Equivalent multivariate normal
>>> from scipy.stats import multivariate_normal
>>> vectorised_X = X.T.flatten()
>>> equiv_mean = M.T.flatten()
>>> equiv_cov = np.kron(V,U)
>>> multivariate_normal.pdf(vectorised_X, mean=equiv_mean, cov=equiv_cov)
0.023410202050005054
```

Methods

`pdf(X, mean=None, rowcov=1, colcov=1)`	Probability density function.
`logpdf(X, mean=None, rowcov=1, colcov=1)`	Log of the probability density function.
`rvs(mean=None, rowcov=1, colcov=1, size=1, random_state=None)`	Draw random samples.

`scipy.stats.`**`dirichlet`** **= <scipy.stats._multivariate.dirichlet_gen object>**

A Dirichlet random variable.

The *`alpha`* keyword specifies the concentration parameters of the distribution.

New in version 0.15.0.

Parameters

x : array_like
    Quantiles, with the last axis of *x* denoting the components.

alpha : array_like
    The concentration parameters. The number of entries determines the dimensionality of the distribution.

random_state : None or int or np.random.RandomState instance, optional
    If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

Alternatively, the object may be called (as a function) to fix concentration parameters, returning a "frozen" Dirichlet random variable:

rv = dirichlet(alpha)

- Frozen object with the same methods but holding the given concentration parameters fixed.

Notes

Each α entry must be positive. The distribution has only support on the simplex defined by

$$\sum_{i=1}^{K} x_i \le 1$$

The probability density function for *dirichlet* is

$$f(x) = \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

where

$$\mathrm{B}(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma\bigl(\sum_{i=1}^{K} \alpha_i\bigr)}$$

and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$, the concentration parameters and K is the dimension of the space where x takes values.

Note that the dirichlet interface is somewhat inconsistent. The array returned by the rvs function is transposed with respect to the format expected by the pdf and logpdf.

Methods

`pdf(x, alpha)`	Probability density function.
`logpdf(x, alpha)`	Log of the probability density function.
`rvs(alpha, size=1, random_state=None)`	Draw random samples from a Dirichlet distribution.
`mean(alpha)`	The mean of the Dirichlet distribution
`var(alpha)`	The variance of the Dirichlet distribution
`entropy(alpha)`	Compute the differential entropy of the Dirichlet distribution.

`scipy.stats.`**`wishart`** **= <scipy.stats._multivariate.wishart_gen object>**

A Wishart random variable.

The *df* keyword specifies the degrees of freedom. The *scale* keyword specifies the scale matrix, which must be symmetric and positive definite. In this context, the scale matrix is often interpreted in terms of a multivariate normal precision matrix (the inverse of the covariance matrix).

Parameters

x : array_like
: Quantiles, with the last axis of *x* denoting the components.

df : int
: Degrees of freedom, must be greater than or equal to dimension of the scale matrix

scale : array_like
: Symmetric positive definite scale matrix of the distribution

random_state : None or int or np.random.RandomState instance, optional
: If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

Alternatively, the object may be called (as a function) to fix the degrees of freedom and scale parameters, returning a "frozen" Wishart random variable:

rv = wishart(df=1, scale=1)

- Frozen object with the same methods but holding the given degrees of freedom and scale fixed.

See also:

invwishart, *chi2*

Notes

The scale matrix *scale* must be a symmetric positive definite matrix. Singular matrices, including the symmetric positive semi-definite case, are not supported.

The Wishart distribution is often denoted

$$W_p(\nu, \Sigma)$$

where ν is the degrees of freedom and Σ is the $p \times p$ scale matrix.

The probability density function for `wishart` has support over positive definite matrices S; if $S \sim W_p(\nu, \Sigma)$, then its PDF is given by:

$$f(S) = \frac{|S|^{\frac{\nu - p - 1}{2}}}{2^{\frac{\nu p}{2}} |\Sigma|^\frac{\nu}{2} \Gamma_p\left(\frac{\nu}{2}\right)} \exp\left(-tr(\Sigma^{-1} S) / 2 \right)$$

If $S \sim W_p(\nu, \Sigma)$ (Wishart) then $S^{-1} \sim W_p^{-1}(\nu, \Sigma^{-1})$ (inverse Wishart).

If the scale matrix is 1-dimensional and equal to one, then the Wishart distribution $W_1(\nu, 1)$ collapses to the $\chi^2(\nu)$ distribution.

New in version 0.16.0.

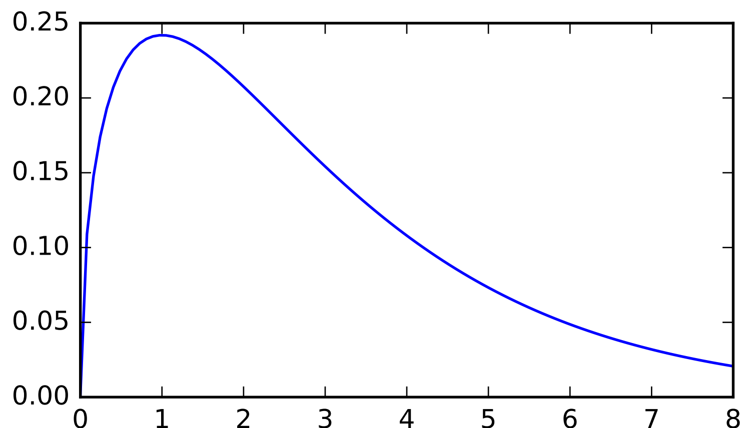
References

[R652], [R653]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy.stats import wishart, chi2
>>> x = np.linspace(1e-5, 8, 100)
>>> w = wishart.pdf(x, df=3, scale=1); w[:5]
array([ 0.00126156,  0.10892176,  0.14793434,  0.17400548,  0.1929669 ])
>>> c = chi2.pdf(x, 3); c[:5]
array([ 0.00126156,  0.10892176,  0.14793434,  0.17400548,  0.1929669 ])
>>> plt.plot(x, w)
```

The input quantiles can be any shape of array, as long as the last axis labels the components.



Methods

`pdf(x, df, scale)`	Probability density function.
`logpdf(x, df, scale)`	Log of the probability density function.
`rvs(df, scale, size=1, random_state=None)`	Draw random samples from a Wishart distribution.
`entropy()`	Compute the differential entropy of the Wishart distribution.

`scipy.stats.`**`invwishart`** **= <scipy.stats._multivariate.invwishart_gen object>**

An inverse Wishart random variable.

The *df* keyword specifies the degrees of freedom. The *scale* keyword specifies the scale matrix, which must be symmetric and positive definite. In this context, the scale matrix is often interpreted in terms of a multivariate normal covariance matrix.

Parameters

x : array_like
: Quantiles, with the last axis of *x* denoting the components.

df : int
: Degrees of freedom, must be greater than or equal to dimension of the scale matrix

scale : array_like
: Symmetric positive definite scale matrix of the distribution

random_state : None or int or np.random.RandomState instance, optional
: If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

Alternatively, the object may be called (as a function) to fix the degrees of freedom and scale parameters, returning a "frozen" inverse Wishart random variable:

rv = invwishart(df=1, scale=1)

- Frozen object with the same methods but holding the given degrees of freedom and scale fixed.

See also:

`wishart`

Notes

The scale matrix *scale* must be a symmetric positive definite matrix. Singular matrices, including the symmetric positive semi-definite case, are not supported.

The inverse Wishart distribution is often denoted

$$W_p^{-1}(\nu, \Psi)$$

where ν is the degrees of freedom and Ψ is the $p \times p$ scale matrix.

The probability density function for *`invwishart`* has support over positive definite matrices S; if $S \sim W_p^{-1}(\nu, \Sigma)$, then its PDF is given by:

$$f(S) = \frac{|\Sigma|^\frac{\nu}{2}}{2^{ \frac{\nu p}{2} } |S|^{\frac{\nu + p + 1}{2}} \Gamma_p \left(\frac{\nu}{2} \right)} \exp\left(-tr(\Sigma S^{-1}) / 2 \right)$$

If $S \sim W_p^{-1}(\nu, \Psi)$ (inverse Wishart) then $S^{-1} \sim W_p(\nu, \Psi^{-1})$ (Wishart).

If the scale matrix is 1-dimensional and equal to one, then the inverse Wishart distribution $W_1(\nu, 1)$ collapses to the inverse Gamma distribution with parameters shape = $\frac{\nu}{2}$ and scale = $\frac{1}{2}$.

New in version 0.16.0.

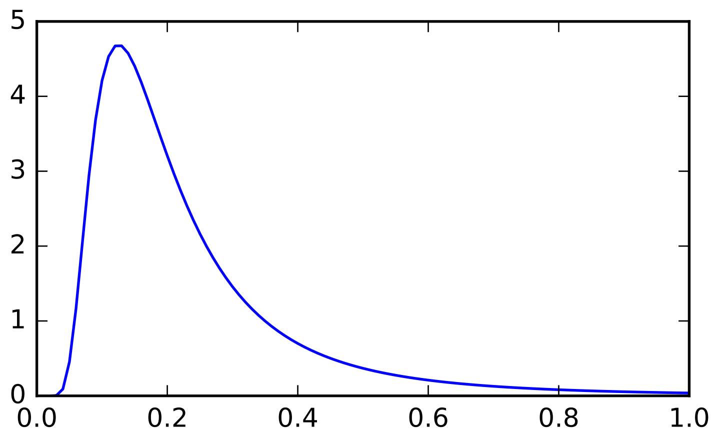
References

[R589], [R590]

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy.stats import invwishart, invgamma
>>> x = np.linspace(0.01, 1, 100)
>>> iw = invwishart.pdf(x, df=6, scale=1)
>>> iw[:3]
array([  1.20546865e-15,   5.42497807e-06,   4.45813929e-03])
>>> ig = invgamma.pdf(x, 6/2., scale=1./2)
>>> ig[:3]
array([  1.20546865e-15,   5.42497807e-06,   4.45813929e-03])
>>> plt.plot(x, iw)
```

The input quantiles can be any shape of array, as long as the last axis labels the components.



Methods

`pdf(x, df, scale)`	Probability density function.
`logpdf(x, df, scale)`	Log of the probability density function.
`rvs(df, scale, size=1, random_state=None)`	Draw random samples from an inverse Wishart distribution.

`scipy.stats.`**`multinomial`** **= <scipy.stats._multivariate.multinomial_gen object>**

A multinomial random variable.

Parameters **x** : array_like

Quantiles, with the last axis of *x* denoting the components.

n : int

Number of trials

p : array_like

Probability of a trial falling into each category; should sum to 1

random_state : None or int or np.random.RandomState instance, optional

If int or RandomState, use it for drawing the random variates. If None (or np.random), the global np.random state is used. Default is None.

See also:

scipy.stats.binom
The binomial distribution.

numpy.random.multinomial
Sampling from the multinomial distribution.

Notes

n should be a positive integer. Each element of *p* should be in the interval $[0, 1]$ and the elements should sum to 1. If they do not sum to 1, the last element of the *p* array is not used and is replaced with the remaining probability left over from the earlier elements.

Alternatively, the object may be called (as a function) to fix the *n* and *p* parameters, returning a "frozen" multinomial random variable:

The probability mass function for `multinomial` is

$$f(x) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k},$$

supported on $x = (x_1, \ldots, x_k)$ where each x_i is a nonnegative integer and their sum is n.

New in version 0.19.0.

Examples

```
>>> from scipy.stats import multinomial
>>> rv = multinomial(8, [0.3, 0.2, 0.5])
>>> rv.pmf([1, 3, 4])
0.042000000000000072
```

The multinomial distribution for $k = 2$ is identical to the corresponding binomial distribution (tiny numerical differences notwithstanding):

```
>>> from scipy.stats import binom
>>> multinomial.pmf([3, 4], n=7, p=[0.4, 0.6])
0.29030399999999973
>>> binom.pmf(3, 7, 0.4)
0.29030400000000012
```

The functions `pmf`, `logpmf`, `entropy`, and `cov` support broadcasting, under the convention that the vector parameters (`x` and `p`) are interpreted as if each row along the last axis is a single object. For instance:

```
>>> multinomial.pmf([[3, 4], [3, 5]], n=[7, 8], p=[.3, .7])
array([0.2268945,  0.25412184])
```

Here, `x.shape == (2, 2)`, `n.shape == (2,)`, and `p.shape == (2,)`, but following the rules mentioned above they behave as if the rows `[3, 4]` and `[3, 5]` in `x` and `[.3, .7]` in `p` were a single object, and as if we had `x.shape = (2,)`, `n.shape = (2,)`, and `p.shape = ()`. To obtain the individual elements without broadcasting, we would do this:

```
>>> multinomial.pmf([3, 4], n=7, p=[.3, .7])
0.2268945
>>> multinomial.pmf([3, 5], 8, p=[.3, .7])
0.25412184
```

This broadcasting also works for `cov`, where the output objects are square matrices of size `p.shape[-1]`. For example:

```
>>> multinomial.cov([4, 5], [[.3, .7], [.4, .6]])
array([[[ 0.84, -0.84],
        [-0.84,  0.84]],
       [[ 1.2 , -1.2 ],
        [-1.2 ,  1.2 ]]])
```

In this example, `n.shape == (2,)` and `p.shape == (2, 2)`, and following the rules above, these broadcast as if `p.shape == (2,)`. Thus the result should also be of shape `(2,)`, but since each output is a 2×2 matrix, the result in fact has shape `(2, 2, 2)`, where `result[0]` is equal to `multinomial.cov(n=4, p=[.3, .7])` and `result[1]` is equal to `multinomial.cov(n=5, p=[.4, .6])`.

Methods

`pmf(x, n, p)`	Probability mass function.
`logpmf(x, n, p)`	Log of the probability mass function.
`rvs(n, p, size=1, random_state=None)`	Draw random samples from a multinomial distribution.
`entropy(n, p)`	Compute the entropy of the multinomial distribution.
`cov(n, p)`	Compute the covariance matrix of the multinomial distribution.

`scipy.stats.`**`special_ortho_group`** **= <scipy.stats._multivariate.special_ortho_group_gen object>**

A matrix-valued SO(N) random variable.

Return a random rotation matrix, drawn from the Haar distribution (the only uniform distribution on SO(n)).

The *dim* keyword specifies the dimension N.

Parameters **dim** : scalar

Dimension of matrices

Notes

This class is wrapping the random_rot code from the MDP Toolkit, https://github.com/mdp-toolkit/mdp-toolkit

Return a random rotation matrix, drawn from the Haar distribution (the only uniform distribution on SO(n)). The algorithm is described in the paper Stewart, G.W., "The efficient generation of random orthogonal matrices with an application to condition estimators", SIAM Journal on Numerical Analysis, 17(3), pp. 403-409, 1980. For more information see http://en.wikipedia.org/wiki/Orthogonal_matrix#Randomization

See also the similar *ortho_group*.

Examples

```
>>> from scipy.stats import special_ortho_group
>>> x = special_ortho_group.rvs(3)
```

```
>>> np.dot(x, x.T)
array([[  1.00000000e+00,   1.13231364e-17,  -2.86852790e-16],
       [  1.13231364e-17,   1.00000000e+00,  -1.46845020e-16],
       [ -2.86852790e-16,  -1.46845020e-16,   1.00000000e+00]])
```

```
>>> import scipy.linalg
>>> scipy.linalg.det(x)
1.0
```

This generates one random matrix from SO(3). It is orthogonal and has a determinant of 1.

Methods

`rvs(dim=None, size=1, random_state=None)`	Draw random samples from SO(N).

`scipy.stats.`**`ortho_group`** **= <scipy.stats._multivariate.ortho_group_gen object>**

A matrix-valued O(N) random variable.

Return a random orthogonal matrix, drawn from the O(N) Haar distribution (the only uniform distribution on O(N)).

The *dim* keyword specifies the dimension N.

Parameters **dim** : scalar

Dimension of matrices

Notes

This class is closely related to *special_ortho_group*.

Some care is taken to avoid numerical error, as per the paper by Mezzadri.

References

[R617]

Examples

```
>>> from scipy.stats import ortho_group
>>> x = ortho_group.rvs(3)
```

```
>>> np.dot(x, x.T)
array([[  1.00000000e+00,   1.13231364e-17,  -2.86852790e-16],
       [  1.13231364e-17,   1.00000000e+00,  -1.46845020e-16],
       [ -2.86852790e-16,  -1.46845020e-16,   1.00000000e+00]])
```

```
>>> import scipy.linalg
>>> np.fabs(scipy.linalg.det(x))
1.0
```

This generates one random matrix from O(3). It is orthogonal and has a determinant of +1 or -1.

Methods

`rvs(dim=None, size=1, random_state=None)`	Draw random samples from O(N).

`scipy.stats.`**`random_correlation`** **= <scipy.stats._multivariate.random_correlation_gen object>**

A random correlation matrix.

Return a random correlation matrix, given a vector of eigenvalues.

The *eigs* keyword specifies the eigenvalues of the correlation matrix, and implies the dimension.

Parameters **eigs** : 1d ndarray

Eigenvalues of correlation matrix.

Notes

Generates a random correlation matrix following a numerically stable algorithm spelled out by Davies & Higham. This algorithm uses a single O(N) similarity transformation to construct a symmetric positive semi-definite matrix, and applies a series of Givens rotations to scale it to have ones on the diagonal.

References

[R628]

Examples

```
>>> from scipy.stats import random_correlation
>>> np.random.seed(514)
>>> x = random_correlation.rvs((.5, .8, 1.2, 1.5))
>>> x
array([[ 1.        , -0.20387311,  0.18366501, -0.04953711],
       [-0.20387311,  1.        , -0.24351129,  0.06703474],
       [ 0.18366501, -0.24351129,  1.        ,  0.38530195],
       [-0.04953711,  0.06703474,  0.38530195,  1.        ]])
```

```
>>> import scipy.linalg
>>> e, v = scipy.linalg.eigh(x)
>>> e
array([ 0.5,  0.8,  1.2,  1.5])
```

Methods

`rvs(eigs=None, random_state=None)`	Draw random correlation matrices, all with eigenvalues eigs.

5.27.3 Discrete distributions

bernoulli	A Bernoulli discrete random variable.
binom	A binomial discrete random variable.
boltzmann	A Boltzmann (Truncated Discrete Exponential) random variable.
dlaplace	A Laplacian discrete random variable.
geom	A geometric discrete random variable.
hypergeom	A hypergeometric discrete random variable.
logser	A Logarithmic (Log-Series, Series) discrete random variable.
nbinom	A negative binomial discrete random variable.
planck	A Planck discrete exponential random variable.
poisson	A Poisson discrete random variable.
randint	A uniform discrete random variable.
skellam	A Skellam discrete random variable.
zipf	A Zipf discrete random variable.

`scipy.stats.`**bernoulli = <scipy.stats._discrete_distns.bernoulli_gen object>**

A Bernoulli discrete random variable.

As an instance of the *rv_discrete* class, *bernoulli* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *bernoulli* is:

```
bernoulli.pmf(k) = 1-p  if k = 0
                 = p    if k = 1
```

for `k` in `{0, 1}`.

bernoulli takes `p` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `bernoulli.pmf(k, p, loc)` is identically equivalent to `bernoulli.pmf(k - loc, p)`.

Examples

```
>>> from scipy.stats import bernoulli
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> p = 0.3
>>> mean, var, skew, kurt = bernoulli.stats(p, moments='mvsk')
```

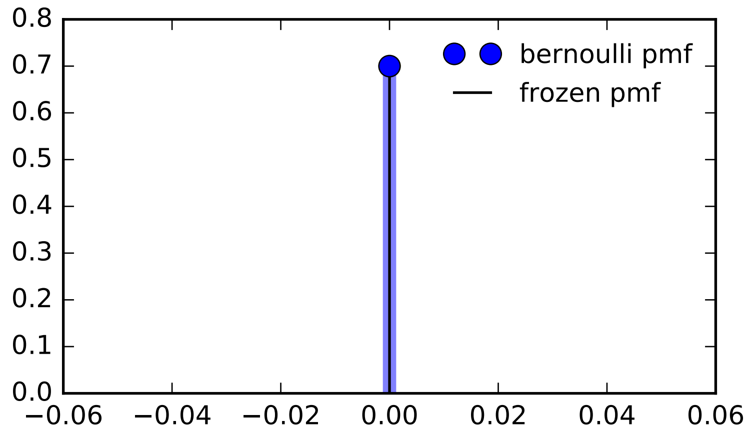
Display the probability mass function (`pmf`):

```
>>> x = np.arange(bernoulli.ppf(0.01, p),
...               bernoulli.ppf(0.99, p))
>>> ax.plot(x, bernoulli.pmf(x, p), 'bo', ms=8, label='bernoulli pmf')
>>> ax.vlines(x, 0, bernoulli.pmf(x, p), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = bernoulli(p)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = bernoulli.cdf(x, p)
>>> np.allclose(x, bernoulli.ppf(prob, p))
True
```

Generate random numbers:

```
>>> r = bernoulli.rvs(p, size=1000)
```

Methods

`rvs(p, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, p, loc=0)`	Probability mass function.
`logpmf(k, p, loc=0)`	Log of the probability mass function.
`cdf(k, p, loc=0)`	Cumulative distribution function.
`logcdf(k, p, loc=0)`	Log of the cumulative distribution function.
`sf(k, p, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, p, loc=0)`	Log of the survival function.
`ppf(q, p, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, p, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(p, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(p, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(p,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(p, loc=0)`	Median of the distribution.
`mean(p, loc=0)`	Mean of the distribution.
`var(p, loc=0)`	Variance of the distribution.
`std(p, loc=0)`	Standard deviation of the distribution.
`interval(alpha, p, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**binom = <scipy.stats._discrete_distns.binom_gen object>**

A binomial discrete random variable.

As an instance of the *rv_discrete* class, *binom* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *binom* is:

```
binom.pmf(k) = choose(n, k) * p**k * (1-p)**(n-k)
```

for `k` in `{0, 1,..., n}`.

binom takes `n` and `p` as shape parameters.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `binom.pmf(k, n, p, loc)` is identically equivalent to `binom.pmf(k - loc, n, p)`.

Examples

```
>>> from scipy.stats import binom
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> n, p = 5, 0.4
>>> mean, var, skew, kurt = binom.stats(n, p, moments='mvsk')
```

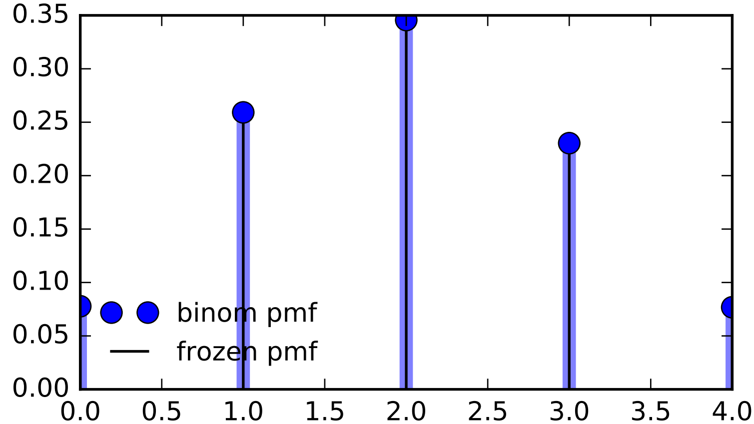
Display the probability mass function (`pmf`):

```
>>> x = np.arange(binom.ppf(0.01, n, p),
...               binom.ppf(0.99, n, p))
>>> ax.plot(x, binom.pmf(x, n, p), 'bo', ms=8, label='binom pmf')
>>> ax.vlines(x, 0, binom.pmf(x, n, p), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = binom(n, p)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = binom.cdf(x, n, p)
>>> np.allclose(x, binom.ppf(prob, n, p))
True
```

Generate random numbers:

```
>>> r = binom.rvs(n, p, size=1000)
```

Methods

`rvs(n, p, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, n, p, loc=0)`	Probability mass function.
`logpmf(k, n, p, loc=0)`	Log of the probability mass function.
`cdf(k, n, p, loc=0)`	Cumulative distribution function.
`logcdf(k, n, p, loc=0)`	Log of the cumulative distribution function.
`sf(k, n, p, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, n, p, loc=0)`	Log of the survival function.
`ppf(q, n, p, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, n, p, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(n, p, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(n, p, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(n, p), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(n, p, loc=0)`	Median of the distribution.
`mean(n, p, loc=0)`	Mean of the distribution.
`var(n, p, loc=0)`	Variance of the distribution.
`std(n, p, loc=0)`	Standard deviation of the distribution.
`interval(alpha, n, p, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`boltzmann`** **= <scipy.stats._discrete_distns.boltzmann_gen object>**

A Boltzmann (Truncated Discrete Exponential) random variable.

As an instance of the *rv_discrete* class, *boltzmann* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *boltzmann* is:

```
boltzmann.pmf(k) = (1-exp(-lambda_)*exp(-lambda_*k)/(1-exp(-lambda_*N))
```

for `k = 0,..., N-1`.

boltzmann takes `lambda_` and `N` as shape parameters.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `boltzmann.pmf(k, lambda_, N, loc)` is identically equivalent to `boltzmann.pmf(k - loc, lambda_, N)`.

Examples

```
>>> from scipy.stats import boltzmann
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> lambda_, N = 1.4, 19
>>> mean, var, skew, kurt = boltzmann.stats(lambda_, N, moments='mvsk')
```

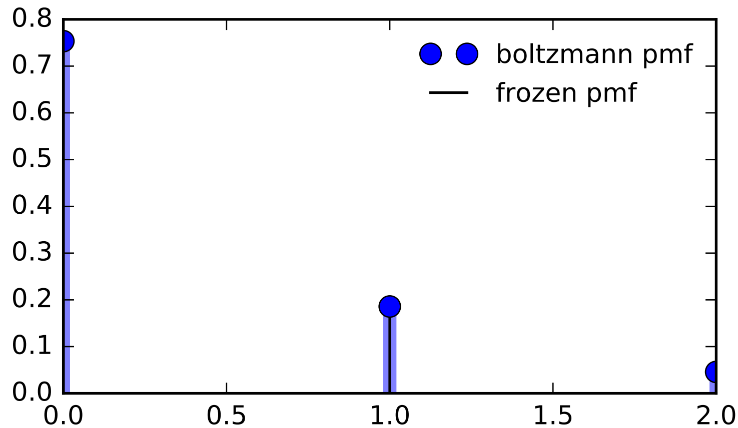
Display the probability mass function (`pmf`):

```
>>> x = np.arange(boltzmann.ppf(0.01, lambda_, N),
...               boltzmann.ppf(0.99, lambda_, N))
>>> ax.plot(x, boltzmann.pmf(x, lambda_, N), 'bo', ms=8, label='boltzmann pmf')
>>> ax.vlines(x, 0, boltzmann.pmf(x, lambda_, N), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = boltzmann(lambda_, N)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = boltzmann.cdf(x, lambda_, N)
>>> np.allclose(x, boltzmann.ppf(prob, lambda_, N))
True
```

Generate random numbers:

```
>>> r = boltzmann.rvs(lambda_, N, size=1000)
```

Methods

`rvs(lambda_, N, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, lambda_, N, loc=0)`	Probability mass function.
`logpmf(k, lambda_, N, loc=0)`	Log of the probability mass function.
`cdf(k, lambda_, N, loc=0)`	Cumulative distribution function.
`logcdf(k, lambda_, N, loc=0)`	Log of the cumulative distribution function.
`sf(k, lambda_, N, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, lambda_, N, loc=0)`	Log of the survival function.
`ppf(q, lambda_, N, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, lambda_, N, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(lambda_, N, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(lambda_, N, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(lambda_, N), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(lambda_, N, loc=0)`	Median of the distribution.
`mean(lambda_, N, loc=0)`	Mean of the distribution.
`var(lambda_, N, loc=0)`	Variance of the distribution.
`std(lambda_, N, loc=0)`	Standard deviation of the distribution.
`interval(alpha, lambda_, N, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`dlaplace`** **= <scipy.stats._discrete_distns.dlaplace_gen object>**

A Laplacian discrete random variable.

As an instance of the *rv_discrete* class, *dlaplace* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *dlaplace* is:

```
dlaplace.pmf(k) = tanh(a/2) * exp(-a*abs(k))
```

for `a > 0`.

dlaplace takes `a` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `dlaplace.pmf(k, a, loc)` is identically equivalent to `dlaplace.pmf(k - loc, a)`.

Examples

```
>>> from scipy.stats import dlaplace
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 0.8
>>> mean, var, skew, kurt = dlaplace.stats(a, moments='mvsk')
```

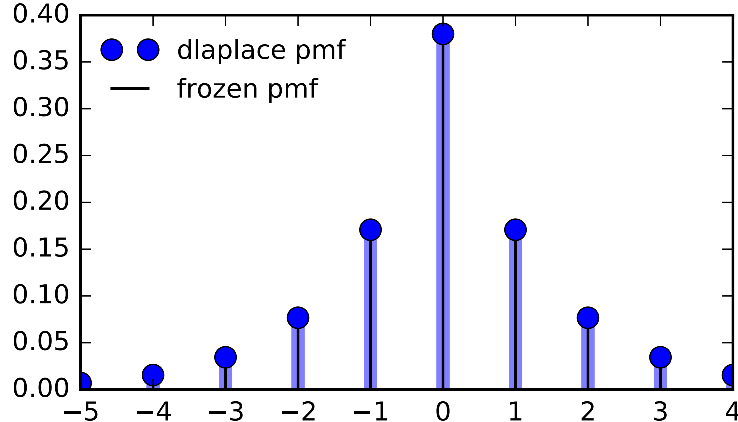
Display the probability mass function (`pmf`):

```
>>> x = np.arange(dlaplace.ppf(0.01, a),
...               dlaplace.ppf(0.99, a))
>>> ax.plot(x, dlaplace.pmf(x, a), 'bo', ms=8, label='dlaplace pmf')
>>> ax.vlines(x, 0, dlaplace.pmf(x, a), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = dlaplace(a)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = dlaplace.cdf(x, a)
>>> np.allclose(x, dlaplace.ppf(prob, a))
True
```

Generate random numbers:

```
>>> r = dlaplace.rvs(a, size=1000)
```

Methods

`rvs(a, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, a, loc=0)`	Probability mass function.
`logpmf(k, a, loc=0)`	Log of the probability mass function.
`cdf(k, a, loc=0)`	Cumulative distribution function.
`logcdf(k, a, loc=0)`	Log of the cumulative distribution function.
`sf(k, a, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, a, loc=0)`	Log of the survival function.
`ppf(q, a, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(a, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(a,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0)`	Median of the distribution.
`mean(a, loc=0)`	Mean of the distribution.
`var(a, loc=0)`	Variance of the distribution.
`std(a, loc=0)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**geom = <scipy.stats._discrete_distns.geom_gen object>**

A geometric discrete random variable.

As an instance of the *rv_discrete* class, *geom* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *geom* is:

```
geom.pmf(k) = (1-p)**(k-1)*p
```

for `k >= 1`.

geom takes `p` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `geom.pmf(k, p, loc)` is identically equivalent to `geom.pmf(k - loc, p)`.

Examples

```
>>> from scipy.stats import geom
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> p = 0.5
>>> mean, var, skew, kurt = geom.stats(p, moments='mvsk')
```

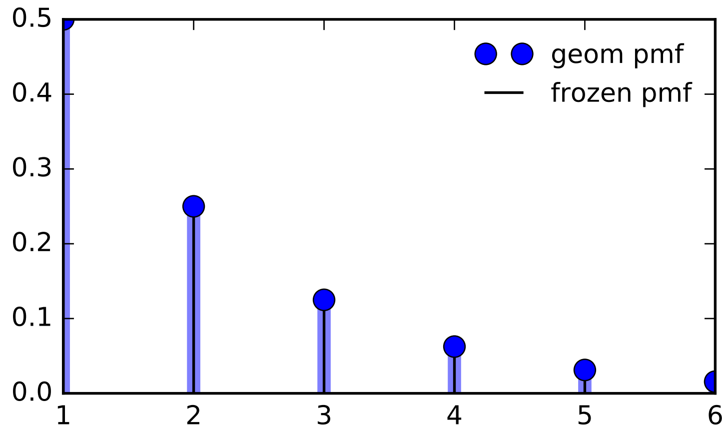
Display the probability mass function (`pmf`):

```
>>> x = np.arange(geom.ppf(0.01, p),
...               geom.ppf(0.99, p))
>>> ax.plot(x, geom.pmf(x, p), 'bo', ms=8, label='geom pmf')
>>> ax.vlines(x, 0, geom.pmf(x, p), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = geom(p)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = geom.cdf(x, p)
>>> np.allclose(x, geom.ppf(prob, p))
True
```

Generate random numbers:

```
>>> r = geom.rvs(p, size=1000)
```

Methods

`rvs(p, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, p, loc=0)`	Probability mass function.
`logpmf(k, p, loc=0)`	Log of the probability mass function.
`cdf(k, p, loc=0)`	Cumulative distribution function.
`logcdf(k, p, loc=0)`	Log of the cumulative distribution function.
`sf(k, p, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, p, loc=0)`	Log of the survival function.
`ppf(q, p, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, p, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(p, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(p, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(p,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(p, loc=0)`	Median of the distribution.
`mean(p, loc=0)`	Mean of the distribution.
`var(p, loc=0)`	Variance of the distribution.
`std(p, loc=0)`	Standard deviation of the distribution.
`interval(alpha, p, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**hypergeom** = <scipy.stats._discrete_distns.hypergeom_gen object>
A hypergeometric discrete random variable.

The hypergeometric distribution models drawing objects from a bin. *M* is the total number of objects, *n* is total number of Type I objects. The random variate represents the number of Type I objects in *N* drawn without replacement from the total population.

As an instance of the `rv_discrete` class, `hypergeom` object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The symbols used to denote the shape parameters (*M*, *n*, and *N*) are not universally accepted. See the Examples for a clarification of the definitions used here.

The probability mass function is defined as,

$$p(k, M, n, N) = \frac{\binom{n}{k} \binom{M - n}{N - k}}{\binom{M}{N}}$$

for $k \in [\max(0, N - M + n), \min(n, N)]$, where the binomial coefficients are defined as,

$$\binom{n}{k} \equiv \frac{n!}{k! (n - k)!}.$$

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `hypergeom.pmf(k, M, n, N, loc)` is identically equivalent to `hypergeom.pmf(k - loc, M, n, N)`.

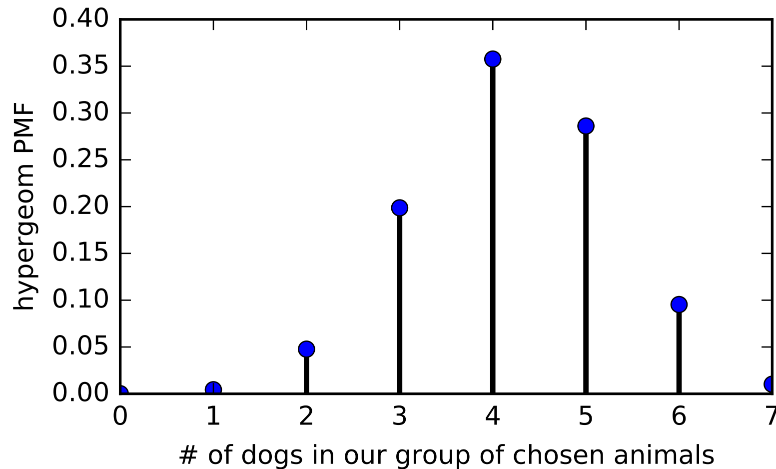
Examples

```
>>> from scipy.stats import hypergeom
>>> import matplotlib.pyplot as plt
```

Suppose we have a collection of 20 animals, of which 7 are dogs. Then if we want to know the probability of finding a given number of dogs if we choose at random 12 of the 20 animals, we can initialize a frozen distribution and plot the probability mass function:

```
>>> [M, n, N] = [20, 7, 12]
>>> rv = hypergeom(M, n, N)
>>> x = np.arange(0, n+1)
>>> pmf_dogs = rv.pmf(x)
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, pmf_dogs, 'bo')
>>> ax.vlines(x, 0, pmf_dogs, lw=2)
>>> ax.set_xlabel('# of dogs in our group of chosen animals')
>>> ax.set_ylabel('hypergeom PMF')
>>> plt.show()
```



Instead of using a frozen distribution we can also use *hypergeom* methods directly. To for example obtain the cumulative distribution function, use:

```
>>> prb = hypergeom.cdf(x, M, n, N)
```

And to generate random numbers:

```
>>> R = hypergeom.rvs(M, n, N, size=10)
```

Methods

`rvs(M, n, N, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, M, n, N, loc=0)`	Probability mass function.
`logpmf(k, M, n, N, loc=0)`	Log of the probability mass function.
`cdf(k, M, n, N, loc=0)`	Cumulative distribution function.
`logcdf(k, M, n, N, loc=0)`	Log of the cumulative distribution function.
`sf(k, M, n, N, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, M, n, N, loc=0)`	Log of the survival function.
`ppf(q, M, n, N, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, M, n, N, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(M, n, N, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(M, n, N, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(M, n, N), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(M, n, N, loc=0)`	Median of the distribution.
`mean(M, n, N, loc=0)`	Mean of the distribution.
`var(M, n, N, loc=0)`	Variance of the distribution.
`std(M, n, N, loc=0)`	Standard deviation of the distribution.
`interval(alpha, M, n, N, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`logser`** **= <scipy.stats._discrete_distns.logser_gen object>**

A Logarithmic (Log-Series, Series) discrete random variable.

As an instance of the *rv_discrete* class, *logser* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *logser* is:

```
logser.pmf(k) = - p**k / (k*log(1-p))
```

for `k >= 1`.

logser takes `p` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `logser.pmf(k, p, loc)` is identically equivalent to `logser.pmf(k - loc, p)`.

Examples

```
>>> from scipy.stats import logser
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> p = 0.6
>>> mean, var, skew, kurt = logser.stats(p, moments='mvsk')
```

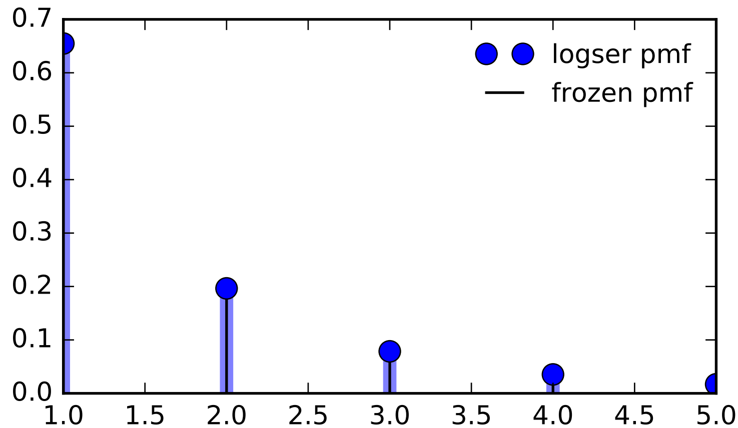
Display the probability mass function (`pmf`):

```
>>> x = np.arange(logser.ppf(0.01, p),
...               logser.ppf(0.99, p))
>>> ax.plot(x, logser.pmf(x, p), 'bo', ms=8, label='logser pmf')
>>> ax.vlines(x, 0, logser.pmf(x, p), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = logser(p)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = logser.cdf(x, p)
>>> np.allclose(x, logser.ppf(prob, p))
True
```

Generate random numbers:

```
>>> r = logser.rvs(p, size=1000)
```

Methods

`rvs(p, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, p, loc=0)`	Probability mass function.
`logpmf(k, p, loc=0)`	Log of the probability mass function.
`cdf(k, p, loc=0)`	Cumulative distribution function.
`logcdf(k, p, loc=0)`	Log of the cumulative distribution function.
`sf(k, p, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, p, loc=0)`	Log of the survival function.
`ppf(q, p, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, p, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(p, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(p, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(p,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(p, loc=0)`	Median of the distribution.
`mean(p, loc=0)`	Mean of the distribution.
`var(p, loc=0)`	Variance of the distribution.
`std(p, loc=0)`	Standard deviation of the distribution.
`interval(alpha, p, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**nbinom = <scipy.stats._discrete_distns.nbinom_gen object>**

A negative binomial discrete random variable.

As an instance of the *rv_discrete* class, *nbinom* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

Negative binomial distribution describes a sequence of i.i.d. Bernoulli trials, repeated until a predefined, non-random number of successes occurs.

The probability mass function of the number of failures for *nbinom* is:

```
nbinom.pmf(k) = choose(k+n-1, n-1) * p**n * (1-p)**k
```

for `k >= 0`.

nbinom takes `n` and `p` as shape parameters where n is the number of successes, whereas p is the probability of a single success.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `nbinom.pmf(k, n, p, loc)` is identically equivalent to `nbinom.pmf(k - loc, n, p)`.

Examples

```
>>> from scipy.stats import nbinom
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> n, p = 0.4, 0.4
>>> mean, var, skew, kurt = nbinom.stats(n, p, moments='mvsk')
```

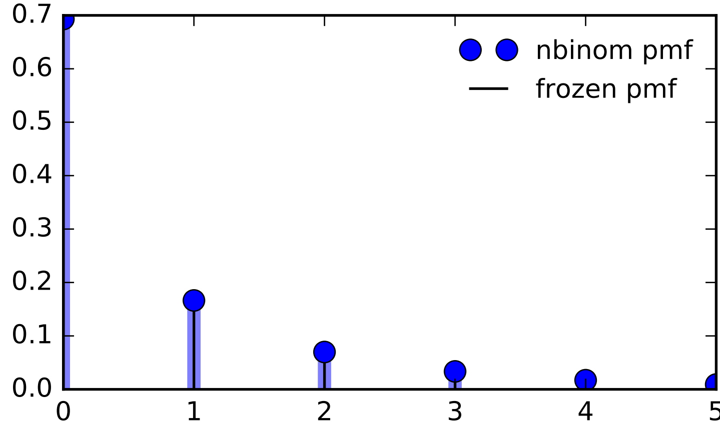
Display the probability mass function (`pmf`):

```
>>> x = np.arange(nbinom.ppf(0.01, n, p),
...               nbinom.ppf(0.99, n, p))
>>> ax.plot(x, nbinom.pmf(x, n, p), 'bo', ms=8, label='nbinom pmf')
>>> ax.vlines(x, 0, nbinom.pmf(x, n, p), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = nbinom(n, p)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = nbinom.cdf(x, n, p)
>>> np.allclose(x, nbinom.ppf(prob, n, p))
True
```

Generate random numbers:

```
>>> r = nbinom.rvs(n, p, size=1000)
```

Methods

`rvs(n, p, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, n, p, loc=0)`	Probability mass function.
`logpmf(k, n, p, loc=0)`	Log of the probability mass function.
`cdf(k, n, p, loc=0)`	Cumulative distribution function.
`logcdf(k, n, p, loc=0)`	Log of the cumulative distribution function.
`sf(k, n, p, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, n, p, loc=0)`	Log of the survival function.
`ppf(q, n, p, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, n, p, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(n, p, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(n, p, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(n, p), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(n, p, loc=0)`	Median of the distribution.
`mean(n, p, loc=0)`	Mean of the distribution.
`var(n, p, loc=0)`	Variance of the distribution.
`std(n, p, loc=0)`	Standard deviation of the distribution.
`interval(alpha, n, p, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**planck** = <scipy.stats._discrete_distns.planck_gen object>

A Planck discrete exponential random variable.

As an instance of the *rv_discrete* class, *planck* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *planck* is:

```
planck.pmf(k) = (1-exp(-lambda_))*exp(-lambda_*k)
```

for `k*lambda_ >= 0`.

planck takes `lambda_` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `planck.pmf(k, lambda_, loc)` is identically equivalent to `planck.pmf(k - loc, lambda_)`.

Examples

```
>>> from scipy.stats import planck
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> lambda_ = 0.51
>>> mean, var, skew, kurt = planck.stats(lambda_, moments='mvsk')
```

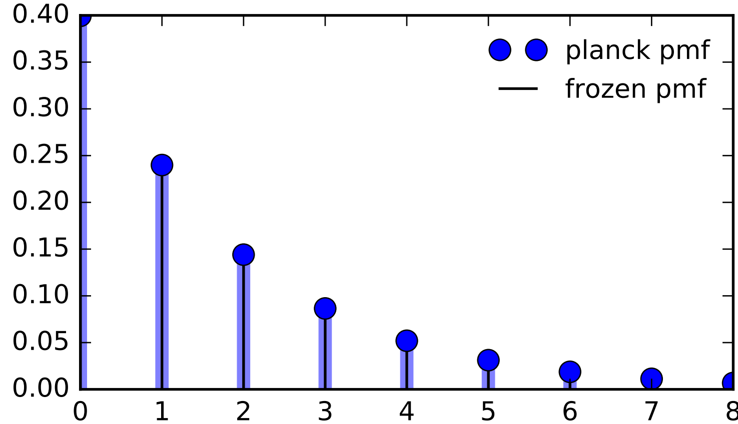
Display the probability mass function (`pmf`):

```
>>> x = np.arange(planck.ppf(0.01, lambda_),
...               planck.ppf(0.99, lambda_))
>>> ax.plot(x, planck.pmf(x, lambda_), 'bo', ms=8, label='planck pmf')
>>> ax.vlines(x, 0, planck.pmf(x, lambda_), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = planck(lambda_)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = planck.cdf(x, lambda_)
>>> np.allclose(x, planck.ppf(prob, lambda_))
True
```

Generate random numbers:

```
>>> r = planck.rvs(lambda_, size=1000)
```

Methods

`rvs(lambda_, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, lambda_, loc=0)`	Probability mass function.
`logpmf(k, lambda_, loc=0)`	Log of the probability mass function.
`cdf(k, lambda_, loc=0)`	Cumulative distribution function.
`logcdf(k, lambda_, loc=0)`	Log of the cumulative distribution function.
`sf(k, lambda_, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, lambda_, loc=0)`	Log of the survival function.
`ppf(q, lambda_, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, lambda_, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(lambda_, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(lambda_, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(lambda_,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(lambda_, loc=0)`	Median of the distribution.
`mean(lambda_, loc=0)`	Mean of the distribution.
`var(lambda_, loc=0)`	Variance of the distribution.
`std(lambda_, loc=0)`	Standard deviation of the distribution.
`interval(alpha, lambda_, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.poisson` = <scipy.stats._discrete_distns.poisson_gen object>

A Poisson discrete random variable.

As an instance of the *rv_discrete* class, *poisson* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *poisson* is:

```
poisson.pmf(k) = exp(-mu) * mu**k / k!
```

for `k >= 0`.

poisson takes `mu` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `poisson.pmf(k, mu, loc)` is identically equivalent to `poisson.pmf(k - loc, mu)`.

Examples

```
>>> from scipy.stats import poisson
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mu = 0.6
>>> mean, var, skew, kurt = poisson.stats(mu, moments='mvsk')
```

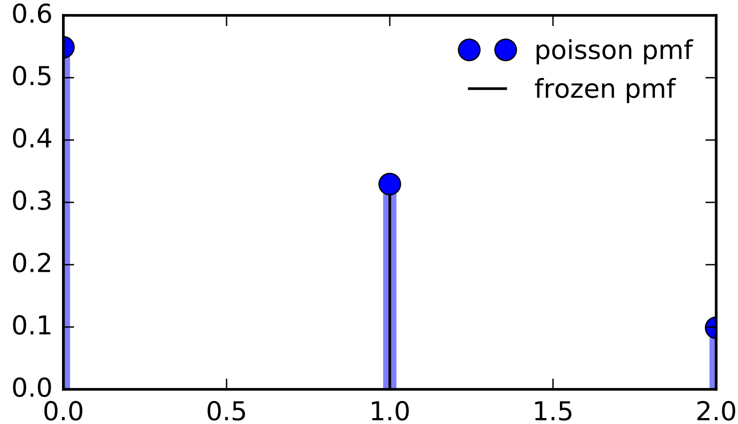
Display the probability mass function (`pmf`):

```
>>> x = np.arange(poisson.ppf(0.01, mu),
...               poisson.ppf(0.99, mu))
>>> ax.plot(x, poisson.pmf(x, mu), 'bo', ms=8, label='poisson pmf')
>>> ax.vlines(x, 0, poisson.pmf(x, mu), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = poisson(mu)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = poisson.cdf(x, mu)
>>> np.allclose(x, poisson.ppf(prob, mu))
True
```

Generate random numbers:

```
>>> r = poisson.rvs(mu, size=1000)
```

Methods

`rvs(mu, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, mu, loc=0)`	Probability mass function.
`logpmf(k, mu, loc=0)`	Log of the probability mass function.
`cdf(k, mu, loc=0)`	Cumulative distribution function.
`logcdf(k, mu, loc=0)`	Log of the cumulative distribution function.
`sf(k, mu, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, mu, loc=0)`	Log of the survival function.
`ppf(q, mu, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, mu, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(mu, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(mu, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(mu,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(mu, loc=0)`	Median of the distribution.
`mean(mu, loc=0)`	Mean of the distribution.
`var(mu, loc=0)`	Variance of the distribution.
`std(mu, loc=0)`	Standard deviation of the distribution.
`interval(alpha, mu, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**randint** = <scipy.stats._discrete_distns.randint_gen object>

A uniform discrete random variable.

As an instance of the *rv_discrete* class, *randint* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *randint* is:

```
randint.pmf(k) = 1./(high - low)
```

for `k = low, ..., high - 1`.

randint takes `low` and `high` as shape parameters.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `randint.pmf(k, low, high, loc)` is identically equivalent to `randint.pmf(k - loc, low, high)`.

Examples

```
>>> from scipy.stats import randint
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> low, high = 7, 31
>>> mean, var, skew, kurt = randint.stats(low, high, moments='mvsk')
```

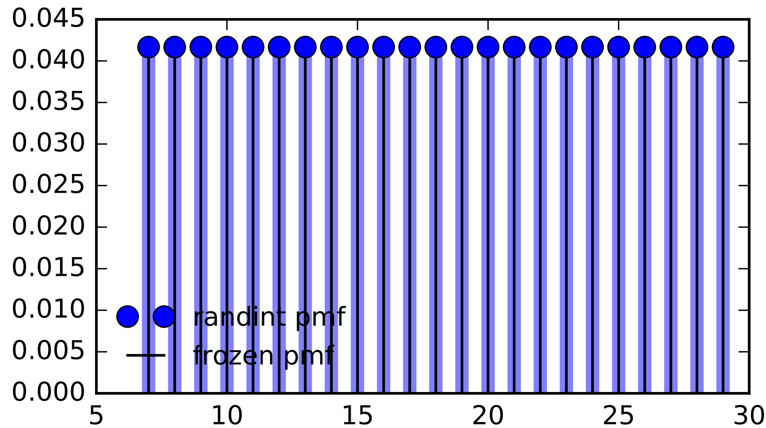
Display the probability mass function (`pmf`):

```
>>> x = np.arange(randint.ppf(0.01, low, high),
...               randint.ppf(0.99, low, high))
>>> ax.plot(x, randint.pmf(x, low, high), 'bo', ms=8, label='randint pmf')
>>> ax.vlines(x, 0, randint.pmf(x, low, high), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = randint(low, high)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = randint.cdf(x, low, high)
>>> np.allclose(x, randint.ppf(prob, low, high))
True
```

Generate random numbers:

```
>>> r = randint.rvs(low, high, size=1000)
```

Methods

`rvs(low, high, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, low, high, loc=0)`	Probability mass function.
`logpmf(k, low, high, loc=0)`	Log of the probability mass function.
`cdf(k, low, high, loc=0)`	Cumulative distribution function.
`logcdf(k, low, high, loc=0)`	Log of the cumulative distribution function.
`sf(k, low, high, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, low, high, loc=0)`	Log of the survival function.
`ppf(q, low, high, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, low, high, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(low, high, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(low, high, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(low, high), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(low, high, loc=0)`	Median of the distribution.
`mean(low, high, loc=0)`	Mean of the distribution.
`var(low, high, loc=0)`	Variance of the distribution.
`std(low, high, loc=0)`	Standard deviation of the distribution.
`interval(alpha, low, high, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

scipy.stats.**skellam** = <scipy.stats._discrete_distns.skellam_gen object>

A Skellam discrete random variable.

As an instance of the *rv_discrete* class, *skellam* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

Probability distribution of the difference of two correlated or uncorrelated Poisson random variables.

Let k1 and k2 be two Poisson-distributed r.v. with expected values lam1 and lam2. Then, `k1 - k2` follows a Skellam distribution with parameters `mu1 = lam1 - rho*sqrt(lam1*lam2)` and `mu2 = lam2 - rho*sqrt(lam1*lam2)`, where rho is the correlation coefficient between k1 and k2. If the two Poisson-distributed r.v. are independent then `rho = 0`.

Parameters mu1 and mu2 must be strictly positive.

For details see: http://en.wikipedia.org/wiki/Skellam_distribution

skellam takes `mu1` and `mu2` as shape parameters.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `skellam.pmf(k, mu1, mu2, loc)` is identically equivalent to `skellam.pmf(k - loc, mu1, mu2)`.

Examples

```
>>> from scipy.stats import skellam
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> mu1, mu2 = 15, 8
>>> mean, var, skew, kurt = skellam.stats(mu1, mu2, moments='mvsk')
```

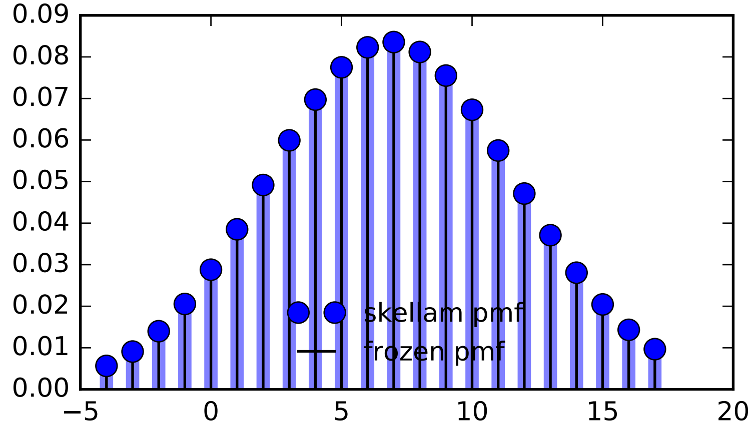
Display the probability mass function (`pmf`):

```
>>> x = np.arange(skellam.ppf(0.01, mu1, mu2),
...               skellam.ppf(0.99, mu1, mu2))
>>> ax.plot(x, skellam.pmf(x, mu1, mu2), 'bo', ms=8, label='skellam pmf')
>>> ax.vlines(x, 0, skellam.pmf(x, mu1, mu2), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = skellam(mu1, mu2)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = skellam.cdf(x, mu1, mu2)
>>> np.allclose(x, skellam.ppf(prob, mu1, mu2))
True
```

Generate random numbers:

```
>>> r = skellam.rvs(mu1, mu2, size=1000)
```

Methods

`rvs(mu1, mu2, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, mu1, mu2, loc=0)`	Probability mass function.
`logpmf(k, mu1, mu2, loc=0)`	Log of the probability mass function.
`cdf(k, mu1, mu2, loc=0)`	Cumulative distribution function.
`logcdf(k, mu1, mu2, loc=0)`	Log of the cumulative distribution function.
`sf(k, mu1, mu2, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, mu1, mu2, loc=0)`	Log of the survival function.
`ppf(q, mu1, mu2, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, mu1, mu2, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(mu1, mu2, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(mu1, mu2, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(mu1, mu2), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(mu1, mu2, loc=0)`	Median of the distribution.
`mean(mu1, mu2, loc=0)`	Mean of the distribution.
`var(mu1, mu2, loc=0)`	Variance of the distribution.
`std(mu1, mu2, loc=0)`	Standard deviation of the distribution.
`interval(alpha, mu1, mu2, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

`scipy.stats.`**`zipf`** **= <scipy.stats._discrete_distns.zipf_gen object>**

A Zipf discrete random variable.

As an instance of the *rv_discrete* class, *zipf* object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

Notes

The probability mass function for *zipf* is:

```
zipf.pmf(k, a) = 1/(zeta(a) * k**a)
```

for `k >= 1`.

zipf takes `a` as shape parameter.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `zipf.pmf(k, a, loc)` is identically equivalent to `zipf.pmf(k - loc, a)`.

Examples

```
>>> from scipy.stats import zipf
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(1, 1)
```

Calculate a few first moments:

```
>>> a = 6.5
>>> mean, var, skew, kurt = zipf.stats(a, moments='mvsk')
```

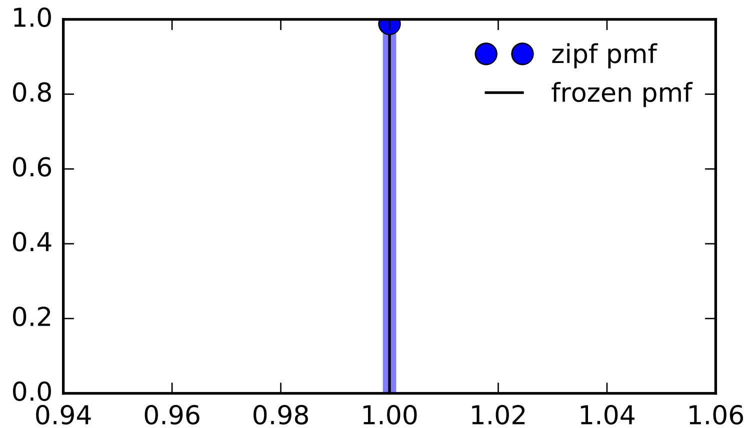
Display the probability mass function (`pmf`):

```
>>> x = np.arange(zipf.ppf(0.01, a),
...               zipf.ppf(0.99, a))
>>> ax.plot(x, zipf.pmf(x, a), 'bo', ms=8, label='zipf pmf')
>>> ax.vlines(x, 0, zipf.pmf(x, a), colors='b', lw=5, alpha=0.5)
```

Alternatively, the distribution object can be called (as a function) to fix the shape and location. This returns a "frozen" RV object holding the given parameters fixed.

Freeze the distribution and display the frozen `pmf`:

```
>>> rv = zipf(a)
>>> ax.vlines(x, 0, rv.pmf(x), colors='k', linestyles='-', lw=1,
...         label='frozen pmf')
>>> ax.legend(loc='best', frameon=False)
>>> plt.show()
```



Check accuracy of `cdf` and `ppf`:

```
>>> prob = zipf.cdf(x, a)
>>> np.allclose(x, zipf.ppf(prob, a))
True
```

Generate random numbers:

```
>>> r = zipf.rvs(a, size=1000)
```

Methods

`rvs(a, loc=0, size=1, random_state=None)`	Random variates.
`pmf(k, a, loc=0)`	Probability mass function.
`logpmf(k, a, loc=0)`	Log of the probability mass function.
`cdf(k, a, loc=0)`	Cumulative distribution function.
`logcdf(k, a, loc=0)`	Log of the cumulative distribution function.
`sf(k, a, loc=0)`	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
`logsf(k, a, loc=0)`	Log of the survival function.
`ppf(q, a, loc=0)`	Percent point function (inverse of `cdf` — percentiles).
`isf(q, a, loc=0)`	Inverse survival function (inverse of `sf`).
`stats(a, loc=0, moments='mv')`	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').
`entropy(a, loc=0)`	(Differential) entropy of the RV.
`expect(func, args=(a,), loc=0, lb=None, ub=None, conditional=False)`	Expected value of a function (of one argument) with respect to the distribution.
`median(a, loc=0)`	Median of the distribution.
`mean(a, loc=0)`	Mean of the distribution.
`var(a, loc=0)`	Variance of the distribution.
`std(a, loc=0)`	Standard deviation of the distribution.
`interval(alpha, a, loc=0)`	Endpoints of the range that contains alpha percent of the distribution

5.27.4 Statistical functions

Several of these functions have a similar version in scipy.stats.mstats which work for masked arrays.

describe(a[, axis, ddof, bias, nan_policy])	Computes several descriptive statistics of the passed array.
gmean(a[, axis, dtype])	Compute the geometric mean along the specified axis.
hmean(a[, axis, dtype])	Calculates the harmonic mean along the specified axis.
kurtosis(a[, axis, fisher, bias, nan_policy])	Computes the kurtosis (Fisher or Pearson) of a dataset.
kurtosistest(a[, axis, nan_policy])	Tests whether a dataset has normal kurtosis
mode(a[, axis, nan_policy])	Returns an array of the modal (most common) value in the passed array.
moment(a[, moment, axis, nan_policy])	Calculates the nth moment about the mean for a sample.
normaltest(a[, axis, nan_policy])	Tests whether a sample differs from a normal distribution.
skew(a[, axis, bias, nan_policy])	Computes the skewness of a data set.
skewtest(a[, axis, nan_policy])	Tests whether the skew is different from the normal distribution.
kstat(data[, n])	Return the nth k-statistic (1<=n<=4 so far).
kstatvar(data[, n])	Returns an unbiased estimator of the variance of the k-statistic.
tmean(a[, limits, inclusive, axis])	Compute the trimmed mean.
tvar(a[, limits, inclusive, axis, ddof])	Compute the trimmed variance
tmin(a[, lowerlimit, axis, inclusive, ...])	Compute the trimmed minimum
tmax(a[, upperlimit, axis, inclusive, ...])	Compute the trimmed maximum
tstd(a[, limits, inclusive, axis, ddof])	Compute the trimmed sample standard deviation
tsem(a[, limits, inclusive, axis, ddof])	Compute the trimmed standard error of the mean.
variation(a[, axis, nan_policy])	Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.
find_repeats(arr)	Find repeats and repeat counts.
trim_mean(a, proportiontocut[, axis])	Return mean of array after trimming distribution from both tails.

scipy.stats.**describe**(*a*, *axis=0*, *ddof=1*, *bias=True*, *nan_policy='propagate'*)

Computes several descriptive statistics of the passed array.

Parameters

a : array_like
: Input data.

axis : int or None, optional
: Axis along which statistics are calculated. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
: Delta degrees of freedom (only for variance). Default is 1.

bias : bool, optional
: If False, then the skewness and kurtosis calculations are corrected for statistical bias.

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

nobs : int
: Number of observations (length of data along *axis*).

minmax: tuple of ndarrays or floats
: Minimum and maximum value of data array.

mean : ndarray or float

Arithmetic mean of data along axis.

variance : ndarray or float
Unbiased variance of the data along axis, denominator is number of observations minus one.

skewness : ndarray or float
Skewness, based on moment calculations with denominator equal to the number of observations, i.e. no degrees of freedom correction.

kurtosis : ndarray or float
Kurtosis (Fisher). The kurtosis is normalized so that it is zero for the normal distribution. No degrees of freedom are used.

See also:

`skew`, `kurtosis`

Examples

```
>>> from scipy import stats
>>> a = np.arange(10)
>>> stats.describe(a)
DescribeResult(nobs=10, minmax=(0, 9), mean=4.5, variance=9.1666666666666661,
               skewness=0.0, kurtosis=-1.2242424242424244)
>>> b = [[1, 2], [3, 4]]
>>> stats.describe(b)
DescribeResult(nobs=2, minmax=(array([1, 2]), array([3, 4])),
               mean=array([ 2.,  3.]), variance=array([ 2.,  2.]),
               skewness=array([ 0.,  0.]), kurtosis=array([-2., -2.]))
```

`scipy.stats.`**`gmean`**(*a*, *axis=0*, *dtype=None*)

Compute the geometric mean along the specified axis.

Returns the geometric average of the array elements. That is: n-th root of (x1 * x2 * ... * xn)

Parameters

a : array_like
Input array or object that can be converted to an array.

axis : int or None, optional
Axis along which the geometric mean is computed. Default is 0. If None, compute over the whole array *a*.

dtype : dtype, optional
Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

Returns

gmean : ndarray
see dtype parameter above

See also:

`numpy.mean`	Arithmetic average
`numpy.average`	Weighted average
`hmean`	Harmonic mean

Notes

The geometric average is computed over a single dimension of the input array, axis=0 by default, or all values in the array if axis=None. float64 intermediate and return values are used for integer inputs.

Use masked arrays to ignore any non-finite values in the input or that arise in the calculations such as Not a Number and infinity because masked arrays automatically mask any non-finite values.

scipy.stats.**hmean**(*a*, *axis=0*, *dtype=None*)

Calculates the harmonic mean along the specified axis.

That is: n / (1/x1 + 1/x2 + ... + 1/xn)

Parameters

a : array_like
: Input array, masked array or object that can be converted to an array.

axis : int or None, optional
: Axis along which the harmonic mean is computed. Default is 0. If None, compute over the whole array *a*.

dtype : dtype, optional
: Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer *dtype* with a precision less than that of the default platform integer. In that case, the default platform integer is used.

Returns

hmean : ndarray
: see *dtype* parameter above

See also:

numpy.mean Arithmetic average

numpy.average Weighted average

gmean Geometric mean

Notes

The harmonic mean is computed over a single dimension of the input array, axis=0 by default, or all values in the array if axis=None. float64 intermediate and return values are used for integer inputs.

Use masked arrays to ignore any non-finite values in the input or that arise in the calculations such as Not a Number and infinity.

scipy.stats.**kurtosis**(*a*, *axis=0*, *fisher=True*, *bias=True*, *nan_policy='propagate'*)

Computes the kurtosis (Fisher or Pearson) of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution.

If bias is False then the kurtosis is calculated using k statistics to eliminate bias coming from biased moment estimators

Use `kurtosistest` to see if result is close enough to normal.

Parameters

a : array
: data for which the kurtosis is calculated

axis : int or None, optional
: Axis along which the kurtosis is calculated. Default is 0. If None, compute over the whole array *a*.

fisher : bool, optional
: If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).

bias : bool, optional
: If False, then the calculations are corrected for statistical bias.

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

kurtosis : array

The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's definition and 0 for Pearson's definition.

References

[R601]

`scipy.stats.kurtosistest(a, axis=0, nan_policy='propagate')`

Tests whether a dataset has normal kurtosis

This function tests the null hypothesis that the kurtosis of the population from which the sample was drawn is that of the normal distribution: `kurtosis = 3(n-1)/(n+1)`.

Parameters

a : array
  array of the sample data

axis : int or None, optional
  Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional
  Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float
  The computed z-score for this test.

pvalue : float
  The 2-sided p-value for the hypothesis test

Notes

Valid only for n>20. The Z-score is set to 0 for bad entries. This function uses the method described in *[R602]*.

References

[R602]

`scipy.stats.mode(a, axis=0, nan_policy='propagate')`

Returns an array of the modal (most common) value in the passed array.

If there is more than one such value, only the smallest is returned. The bin-count for the modal bins is also returned.

Parameters

a : array_like
  n-dimensional array of which to find mode(s).

axis : int or None, optional
  Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional
  Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

mode : ndarray
  Array of modal values.

count : ndarray
  Array of counts for each mode.

Examples

```
>>> a = np.array([[6, 8, 3, 0],
...               [3, 2, 1, 7],
...               [8, 1, 8, 4],
```

```
...                 [5, 3, 0, 5],
...                 [4, 7, 5, 9]])
>>> from scipy import stats
>>> stats.mode(a)
(array([[3, 1, 0, 0]]), array([[1, 1, 1, 1]]))
```

To get mode of whole array, specify `axis=None`:

```
>>> stats.mode(a, axis=None)
(array([3]), array([3]))
```

`scipy.stats.moment(a, moment=1, axis=0, nan_policy='propagate')`

Calculates the nth moment about the mean for a sample.

A moment is a specific quantitative measure of the shape of a set of points. It is often used to calculate coefficients of skewness and kurtosis due to its close relationship with them.

Parameters

a : array_like
: data

moment : int or array_like of ints, optional
: order of central moment that is returned. Default is 1.

axis : int or None, optional
: Axis along which the central moment is computed. Default is 0. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

n-th central moment : ndarray or float
: The appropriate moment along the given axis or over all values if axis is None. The denominator for the moment calculation is the number of observations, no degrees of freedom correction is done.

See also:

kurtosis, *skew*, *describe*

Notes

The k-th central moment of a data sample is:

$$m_k = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^k$$

Where n is the number of samples and x-bar is the mean. This function uses exponentiation by squares *[R609]* for efficiency.

References

[R609]

`scipy.stats.normaltest(a, axis=0, nan_policy='propagate')`

Tests whether a sample differs from a normal distribution.

This function tests the null hypothesis that a sample comes from a normal distribution. It is based on D'Agostino and Pearson's *[R614]*, *[R615]* test that combines skew and kurtosis to produce an omnibus test of normality.

Parameters

a : array_like
: The array containing the data to be tested.

axis : int or None, optional

Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float or array
`s^2 + k^2`, where `s` is the z-score returned by *skewtest* and `k` is the z-score returned by *kurtosistest*.

pvalue : float or array
A 2-sided chi squared probability for the hypothesis test.

References

[R614], [R615]

`scipy.stats.skew`(*a*, *axis=0*, *bias=True*, *nan_policy='propagate'*)

Computes the skewness of a data set.

For normally distributed data, the skewness should be about 0. A skewness value > 0 means that there is more weight in the left tail of the distribution. The function *skewtest* can be used to determine if the skewness value is close enough to 0, statistically speaking.

Parameters

a : ndarray
data

axis : int or None, optional
Axis along which skewness is calculated. Default is 0. If None, compute over the whole array *a*.

bias : bool, optional
If False, then the calculations are corrected for statistical bias.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

skewness : ndarray
The skewness of values along an axis, returning 0 where all values are equal.

References

[R635]

`scipy.stats.skewtest`(*a*, *axis=0*, *nan_policy='propagate'*)

Tests whether the skew is different from the normal distribution.

This function tests the null hypothesis that the skewness of the population that the sample was drawn from is the same as that of a corresponding normal distribution.

Parameters

a : array
The data to be tested

axis : int or None, optional
Axis along which statistics are calculated. Default is 0. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float
The computed z-score for this test.

pvalue : float
a 2-sided p-value for the hypothesis test

Notes

The sample size must be at least 8.

References

[R637]

`scipy.stats.kstat(data, n=2)`

Return the nth k-statistic (1<=n<=4 so far).

The nth k-statistic k_n is the unique symmetric unbiased estimator of the nth cumulant kappa_n.

Parameters
- **data** : array_like
  Input array. Note that n-D input gets flattened.
- **n** : int, {1, 2, 3, 4}, optional
  Default is equal to 2.

Returns
- **kstat** : float
  The nth k-statistic.

See also:

kstatvar Returns an unbiased estimator of the variance of the k-statistic.
moment Returns the n-th central moment about the mean for a sample.

Notes

For a sample size n, the first few k-statistics are given by:

$$k_1 = \mu k_2 = \frac{n}{n-1} m_2 k_3 = \frac{n^2}{(n-1)(n-2)} m_3 k_4 = \frac{n^2[(n+1)m_4 - 3(n-1)m_2^2]}{(n-1)(n-2)(n-3)}$$

where μ is the sample mean, m_2 is the sample variance, and m_i is the i-th sample central moment.

References

http://mathworld.wolfram.com/k-Statistic.html

http://mathworld.wolfram.com/Cumulant.html

Examples

```
>>> from scipy import stats
>>> rndm = np.random.RandomState(1234)
```

As sample size increases, n-th moment and n-th k-statistic converge to the same number (although they aren't identical). In the case of the normal distribution, they converge to zero.

```
>>> for n in [2, 3, 4, 5, 6, 7]:
...     x = rndm.normal(size=10**n)
...     m, k = stats.moment(x, 3), stats.kstat(x, 3)
...     print("%.3g %.3g %.3g" % (m, k, m-k))
-0.631 -0.651 0.0194
0.0282 0.0283 -8.49e-05
-0.0454 -0.0454 1.36e-05
7.53e-05 7.53e-05 -2.26e-09
0.00166 0.00166 -4.99e-09
-2.88e-06 -2.88e-06 8.63e-13
```

`scipy.stats.kstatvar(data, n=2)`

Returns an unbiased estimator of the variance of the k-statistic.

See `kstat` for more details of the k-statistic.

Parameters **data** : array_like
Input array. Note that n-D input gets flattened.

n : int, {1, 2}, optional
Default is equal to 2.

Returns **kstatvar** : float
The nth k-statistic variance.

See also:

kstat Returns the n-th k-statistic.
moment Returns the n-th central moment about the mean for a sample.

Notes

The variances of the first few k-statistics are given by:

$$var(k_1) = \frac{\kappa^2}{n} var(k_2) = \frac{\kappa^4}{n} + \frac{2\kappa^2_2}{n-1} var(k_3) = \frac{\kappa^6}{n} + \frac{9\kappa_2\kappa_4}{n-1} + \frac{9\kappa^2_3}{n-1} + \frac{6n\kappa^3_2}{(n-1)(n-2)} var(k_4) = \frac{\kappa^8}{n} + \frac{16\kappa_2\kappa_6}{n-1} + \frac{48\kappa_3\kappa_5}{n-1}$$

scipy.stats.**tmean**(*a*, *limits=None*, *inclusive=(True, True)*, *axis=None*)
Compute the trimmed mean.

This function finds the arithmetic mean of given values, ignoring values outside the given *limits*.

Parameters **a** : array_like
Array of values.

limits : None or (lower limit, upper limit), optional
Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None (default), then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval.

inclusive : (bool, bool), optional
A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
Axis along which to compute test. Default is None.

Returns **tmean** : float

See also:

trim_mean returns mean after trimming a proportion from both tails.

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.tmean(x)
9.5
>>> stats.tmean(x, (3,17))
10.0
```

scipy.stats.**tvar**(*a*, *limits=None*, *inclusive=(True, True)*, *axis=0*, *ddof=1*)
Compute the trimmed variance

This function computes the sample variance of an array of values, while ignoring values which are outside of given *limits*.

Parameters **a** : array_like
Array of values.

limits : None or (lower limit, upper limit), optional
Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval. The default value is None.

inclusive : (bool, bool), optional
A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
Delta degrees of freedom. Default is 1.

Returns

tvar : float
Trimmed variance.

Notes

`tvar` computes the unbiased sample variance, i.e. it uses a correction factor `n / (n - 1)`.

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.tvar(x)
35.0
>>> stats.tvar(x, (3,17))
20.0
```

`scipy.stats.`**`tmin`**(*a*, *lowerlimit=None*, *axis=0*, *inclusive=True*, *nan_policy='propagate'*)

Compute the trimmed minimum

This function finds the miminum value of an array *a* along the specified axis, but only considering values greater than a specified lower limit.

Parameters

a : array_like
array of values

lowerlimit : None or float, optional
Values in the input array less than the given limit will be ignored. When lowerlimit is None, then all values are used. The default value is None.

axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

inclusive : {True, False}, optional
This flag determines whether values exactly equal to the lower limit are included. The default value is True.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

tmin : float, int or ndarray

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
```

```
>>> stats.tmin(x)
0
```

```
>>> stats.tmin(x, 13)
13
```

```
>>> stats.tmin(x, 13, inclusive=False)
14
```

`scipy.stats.tmax`(*a*, *upperlimit=None*, *axis=0*, *inclusive=True*, *nan_policy='propagate'*)

Compute the trimmed maximum

This function computes the maximum value of an array along a given axis, while ignoring values larger than a specified upper limit.

Parameters **a** : array_like

array of values

upperlimit : None or float, optional

Values in the input array greater than the given limit will be ignored. When upperlimit is None, then all values are used. The default value is None.

axis : int or None, optional

Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

inclusive : {True, False}, optional

This flag determines whether values exactly equal to the upper limit are included. The default value is True.

nan_policy : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns **tmax** : float, int or ndarray

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.tmax(x)
19
```

```
>>> stats.tmax(x, 13)
13
```

```
>>> stats.tmax(x, 13, inclusive=False)
12
```

`scipy.stats.tstd`(*a*, *limits=None*, *inclusive=(True, True)*, *axis=0*, *ddof=1*)

Compute the trimmed sample standard deviation

This function finds the sample standard deviation of given values, ignoring values outside the given *limits*.

Parameters **a** : array_like

array of values

limits : None or (lower limit, upper limit), optional

Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None, then all values are used. Either

of the limit values in the tuple can also be None representing a half-open interval. The default value is None.

inclusive : (bool, bool), optional
A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
Delta degrees of freedom. Default is 1.

Returns **tstd** : float

Notes

`tstd` computes the unbiased sample standard deviation, i.e. it uses a correction factor `n / (n - 1)`.

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.tstd(x)
5.9160797830996161
>>> stats.tstd(x, (3,17))
4.4721359549995796
```

`scipy.stats.tsem`(*a*, *limits=None*, *inclusive=(True, True)*, *axis=0*, *ddof=1*)
Compute the trimmed standard error of the mean.

This function finds the standard error of the mean for given values, ignoring values outside the given *limits*.

Parameters **a** : array_like
array of values

limits : None or (lower limit, upper limit), optional
Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval. The default value is None.

inclusive : (bool, bool), optional
A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
Delta degrees of freedom. Default is 1.

Returns **tsem** : float

Notes

`tsem` uses unbiased sample standard deviation, i.e. it uses a correction factor `n / (n - 1)`.

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.tsem(x)
```

```
1.3228756555322954
>>> stats.tsem(x, (3,17))
1.1547005383792515
```

`scipy.stats.variation`(*a*, *axis=0*, *nan_policy='propagate'*)
Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.

Parameters **a** : array_like
Input array.
axis : int or None, optional
Axis along which to calculate the coefficient of variation. Default is 0. If None, compute over the whole array *a*.
nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns **variation** : ndarray
The calculated variation along the requested axis.

References

[R647]

`scipy.stats.find_repeats`(*arr*)
Find repeats and repeat counts.

Parameters **arr** : array_like
Input array. This is cast to float64.

Returns **values** : ndarray
The unique values from the (flattened) input that are repeated.
counts : ndarray
Number of times the corresponding 'value' is repeated.

Notes

In numpy >= 1.9 `numpy.unique` provides similar functionality. The main difference is that *`find_repeats`* only returns repeated values.

Examples

```
>>> from scipy import stats
>>> stats.find_repeats([2, 1, 2, 3, 2, 2, 5])
RepeatedResults(values=array([ 2.]), counts=array([4]))
```

```
>>> stats.find_repeats([[10, 20, 1, 2], [5, 5, 4, 4]])
RepeatedResults(values=array([ 4., 5.]), counts=array([2, 2]))
```

`scipy.stats.trim_mean`(*a*, *proportiontocut*, *axis=0*)
Return mean of array after trimming distribution from both tails.

If *proportiontocut* = 0.1, slices off 'leftmost' and 'rightmost' 10% of scores. The input is sorted before slicing. Slices off less if proportion results in a non-integer slice index (i.e., conservatively slices off *proportiontocut*).

Parameters **a** : array_like
Input array
proportiontocut : float
Fraction to cut off of both tails of the distribution
axis : int or None, optional

Axis along which the trimmed means are computed. Default is 0. If None, compute over the whole array *a*.

Returns **trim_mean** : ndarray

Mean of trimmed array.

See also:

`trimboth`

`tmean` compute the trimmed mean ignoring values outside given *limits*.

Examples

```
>>> from scipy import stats
>>> x = np.arange(20)
>>> stats.trim_mean(x, 0.1)
9.5
>>> x2 = x.reshape(5, 4)
>>> x2
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> stats.trim_mean(x2, 0.25)
array([  8.,   9.,  10.,  11.])
>>> stats.trim_mean(x2, 0.25, axis=1)
array([  1.5,   5.5,   9.5,  13.5,  17.5])
```

`cumfreq`(a[, numbins, defaultreallimits, weights])	Returns a cumulative frequency histogram, using the histogram function.
`histogram2`(*args, **kwds)	`histogram2` is deprecated!
`histogram`(*args, **kwds)	`histogram` is deprecated!
`itemfreq`(a)	Returns a 2-D array of item frequencies.
`percentileofscore`(a, score[, kind])	The percentile rank of a score relative to a list of scores.
`scoreatpercentile`(a, per[, limit, ...])	Calculate the score at a given percentile of the input sequence.
`relfreq`(a[, numbins, defaultreallimits, weights])	Returns a relative frequency histogram, using the histogram function.

`scipy.stats.`**`cumfreq`**(*a*, *numbins=10*, *defaultreallimits=None*, *weights=None*)

Returns a cumulative frequency histogram, using the histogram function.

A cumulative histogram is a mapping that counts the cumulative number of observations in all of the bins up to the specified bin.

Parameters **a** : array_like

Input array.

numbins : int, optional

The number of bins to use for the histogram. Default is 10.

defaultreallimits : tuple (lower, upper), optional

The lower and upper values for the range of the histogram. If no value is given, a range slightly larger than the range of the values in *a* is used. Specifically `(a.min() - s, a.max() + s)`, where `s = (1/2)(a.max() - a.min()) / (numbins - 1)`.

weights : array_like, optional

The weights for each value in *a*. Default is None, which gives each value a weight of 1.0

Returns **cumcount** : ndarray

Binned values of cumulative frequency.

lowerlimit : float

Lower real limit

binsize : float

Width of each bin.

extrapoints : int

Extra points.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> x = [1, 4, 2, 1, 3, 1]
>>> res = stats.cumfreq(x, numbins=4, defaultreallimits=(1.5, 5))
>>> res.cumcount
array([ 1.,  2.,  3.,  3.])
>>> res.extrapoints
3
```

Create a normal distribution with 1000 random values

```
>>> rng = np.random.RandomState(seed=12345)
>>> samples = stats.norm.rvs(size=1000, random_state=rng)
```

Calculate cumulative frequencies

```
>>> res = stats.cumfreq(samples, numbins=25)
```

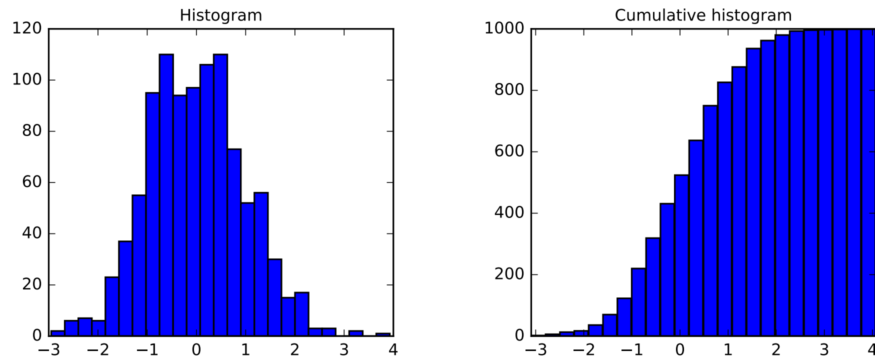
Calculate space of values for x

```
>>> x = res.lowerlimit + np.linspace(0, res.binsize*res.cumcount.size,
...                                  res.cumcount.size)
```

Plot histogram and cumulative histogram

```
>>> fig = plt.figure(figsize=(10, 4))
>>> ax1 = fig.add_subplot(1, 2, 1)
>>> ax2 = fig.add_subplot(1, 2, 2)
>>> ax1.hist(samples, bins=25)
>>> ax1.set_title('Histogram')
>>> ax2.bar(x, res.cumcount, width=res.binsize)
>>> ax2.set_title('Cumulative histogram')
>>> ax2.set_xlim([x.min(), x.max()])
```

```
>>> plt.show()
```



`scipy.stats.`**`histogram2`**`(*args, **kwds)`

`histogram2` is deprecated! scipy.stats.histogram2 is deprecated in scipy 0.16.0; use np.histogram2d instead

Compute histogram using divisions in bins.

Count the number of times values from array *a* fall into numerical ranges defined by *bins*. Range x is given by bins[x] <= range_x < bins[x+1] where x =0,N and N is the length of the *bins* array. The last range is given by bins[N] <= range_N < infinity. Values less than bins[0] are not included in the histogram.

Parameters **a** : array_like of rank 1

The array of values to be assigned into bins

bins [array_like of rank 1] Defines the ranges of values to use during histogramming.

Returns **histogram2** : ndarray of rank 1

Each value represents the occurrences for a given bin (range) of values.

`scipy.stats.`**`histogram`**`(*args, **kwds)`

`histogram` is deprecated! scipy.stats.histogram is deprecated in scipy 0.17.0; use np.histogram instead

`scipy.stats.`**`itemfreq`**`(a)`

Returns a 2-D array of item frequencies.

Parameters **a** : (N,) array_like

Input array.

Returns **itemfreq** : (K, 2) ndarray

A 2-D frequency table. Column 1 contains sorted, unique values from *a*, column 2 contains their respective counts.

Examples

```
>>> from scipy import stats
>>> a = np.array([1, 1, 5, 0, 1, 2, 2, 0, 1, 4])
>>> stats.itemfreq(a)
array([[ 0.,  2.],
       [ 1.,  4.],
       [ 2.,  2.],
       [ 4.,  1.],
       [ 5.,  1.]])
>>> np.bincount(a)
array([2, 4, 2, 0, 1, 1])
```

```
>>> stats.itemfreq(a/10.)
array([[ 0. ,  2. ],
       [ 0.1,  4. ],
       [ 0.2,  2. ],
       [ 0.4,  1. ],
       [ 0.5,  1. ]])
```

`scipy.stats.percentileofscore`(*a*, *score*, *kind='rank'*)

The percentile rank of a score relative to a list of scores.

A `percentileofscore` of, for example, 80% means that 80% of the scores in *a* are below the given score. In the case of gaps or ties, the exact definition depends on the optional keyword, *kind*.

Parameters

a : array_like
: Array of scores to which *score* is compared.

score : int or float
: Score that is compared to the elements in *a*.

kind : {'rank', 'weak', 'strict', 'mean'}, optional
: This optional parameter specifies the interpretation of the resulting score:
  - *"rank": Average percentage ranking of score. In case of* multiple matches, average the percentage rankings of all matching scores.
  - *"weak": This kind corresponds to the definition of a cumulative* distribution function. A percentileofscore of 80% means that 80% of values are less than or equal to the provided score.
  - *"strict": Similar to "weak", except that only values that are* strictly less than the given score are counted.
  - *"mean": The average of the "weak" and "strict" scores, often used in* testing. See http://en.wikipedia.org/wiki/Percentile_rank

Returns

pcos : float
: Percentile-position of score (0-100) relative to *a*.

See also:

`numpy.percentile`

Examples

Three-quarters of the given values lie below a given score:

```
>>> from scipy import stats
>>> stats.percentileofscore([1, 2, 3, 4], 3)
75.0
```

With multiple matches, note how the scores of the two matches, 0.6 and 0.8 respectively, are averaged:

```
>>> stats.percentileofscore([1, 2, 3, 3, 4], 3)
70.0
```

Only 2/5 values are strictly less than 3:

```
>>> stats.percentileofscore([1, 2, 3, 3, 4], 3, kind='strict')
40.0
```

But 4/5 values are less than or equal to 3:

```
>>> stats.percentileofscore([1, 2, 3, 3, 4], 3, kind='weak')
80.0
```

The average between the weak and the strict scores is

```
>>> stats.percentileofscore([1, 2, 3, 3, 4], 3, kind='mean')
60.0
```

`scipy.stats.`**`scoreatpercentile`**(*a*, *per*, *limit=()*, *interpolation_method='fraction'*, *axis=None*)

Calculate the score at a given percentile of the input sequence.

For example, the score at *per=50* is the median. If the desired quantile lies between two data points, we interpolate between them, according to the value of *interpolation*. If the parameter *limit* is provided, it should be a tuple (lower, upper) of two values.

Parameters

a : array_like

A 1-D array of values from which to extract score.

per : array_like

Percentile(s) at which to extract score. Values should be in range [0,100].

limit : tuple, optional

Tuple of two scalars, the lower and upper limits within which to compute the percentile. Values of *a* outside this (closed) interval will be ignored.

interpolation_method : {'fraction', 'lower', 'higher'}, optional

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*

- fraction: `i + (j - i) * fraction` where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- lower: `i`.
- higher: `j`.

axis : int, optional

Axis along which the percentiles are computed. Default is None. If None, compute over the whole array *a*.

Returns

score : float or ndarray

Score at percentile(s).

See also:

`percentileofscore`, `numpy.percentile`

Notes

This function will become obsolete in the future. For Numpy 1.9 and higher, `numpy.percentile` provides all the functionality that *`scoreatpercentile`* provides. And it's significantly faster. Therefore it's recommended to use `numpy.percentile` for users that have numpy >= 1.9.

Examples

```
>>> from scipy import stats
>>> a = np.arange(100)
>>> stats.scoreatpercentile(a, 50)
49.5
```

`scipy.stats.`**`relfreq`**(*a*, *numbins=10*, *defaultreallimits=None*, *weights=None*)

Returns a relative frequency histogram, using the histogram function.

A relative frequency histogram is a mapping of the number of observations in each of the bins relative to the total of observations.

Parameters

a : array_like
: Input array.

numbins : int, optional
: The number of bins to use for the histogram. Default is 10.

defaultreallimits : tuple (lower, upper), optional
: The lower and upper values for the range of the histogram. If no value is given, a range slightly larger than the range of the values in a is used. Specifically `(a.min() - s, a.max() + s)`, where `s = (1/2)(a.max() - a.min()) / (numbins - 1)`.

weights : array_like, optional
: The weights for each value in *a*. Default is None, which gives each value a weight of 1.0

Returns

frequency : ndarray
: Binned values of relative frequency.

lowerlimit : float
: Lower real limit

binsize : float
: Width of each bin.

extrapoints : int
: Extra points.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> a = np.array([2, 4, 1, 2, 3, 2])
>>> res = stats.relfreq(a, numbins=4)
>>> res.frequency
array([ 0.16666667, 0.5       , 0.16666667,  0.16666667])
>>> np.sum(res.frequency)  # relative frequencies should add up to 1
1.0
```

Create a normal distribution with 1000 random values

```
>>> rng = np.random.RandomState(seed=12345)
>>> samples = stats.norm.rvs(size=1000, random_state=rng)
```

Calculate relative frequencies

```
>>> res = stats.relfreq(samples, numbins=25)
```

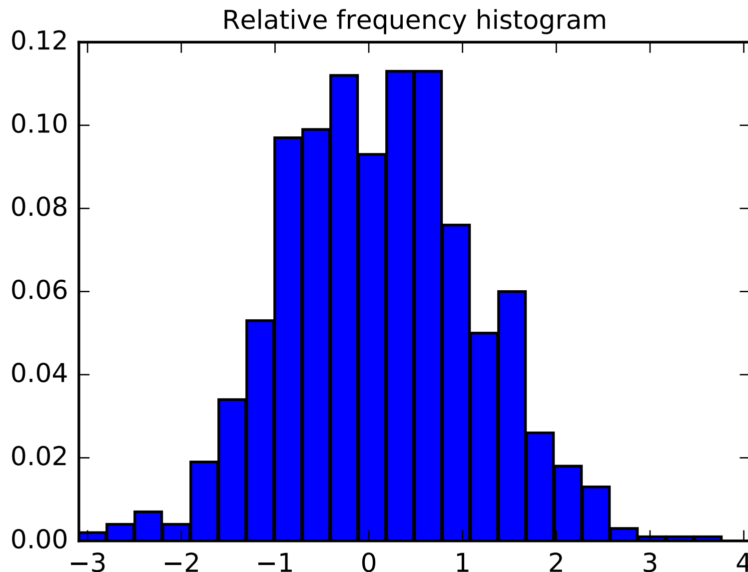
Calculate space of values for x

```
>>> x = res.lowerlimit + np.linspace(0, res.binsize*res.frequency.size,
...                                  res.frequency.size)
```

Plot relative frequency histogram

```
>>> fig = plt.figure(figsize=(5, 4))
>>> ax = fig.add_subplot(1, 1, 1)
>>> ax.bar(x, res.frequency, width=res.binsize)
>>> ax.set_title('Relative frequency histogram')
>>> ax.set_xlim([x.min(), x.max()])
```

```
>>> plt.show()
```



binned_statistic(x, values[, statistic, ...])	Compute a binned statistic for one or more sets of data.
binned_statistic_2d(x, y, values[, ...])	Compute a bidimensional binned statistic for one or more sets of data.
binned_statistic_dd(sample, values[, ...])	Compute a multidimensional binned statistic for a set of data.

scipy.stats.**binned_statistic**(*x*, *values*, *statistic='mean'*, *bins=10*, *range=None*)

Compute a binned statistic for one or more sets of data.

This is a generalization of a histogram function. A histogram divides the space into bins, and returns the count of the number of points in each bin. This function allows the computation of the sum, mean, median, or other statistic of the values (or set of values) within each bin.

Parameters

x : (N,) array_like

A sequence of values to be binned.

values : (N,) array_like or list of (N,) array_like

The data on which the statistic will be computed. This must be the same shape as *x*, or a set of sequences - each the same shape as *x*. If *values* is a set of sequences, the statistic will be computed on each independently.

statistic : string or callable, optional

The statistic to compute (default is 'mean'). The following statistics are available:

- 'mean' : compute the mean of values for points within each bin. Empty bins will be represented by NaN.
- 'median' : compute the median of values for points within each bin. Empty bins will be represented by NaN.
- 'count' : compute the count of points within each bin. This is identical to an unweighted histogram. *values* array is not referenced.
- 'sum' : compute the sum of values for points within each bin. This is identical to a weighted histogram.

- 'min' : compute the minimum of values for points within each bin. Empty bins will be represented by NaN.
- 'max' : compute the maximum of values for point within each bin. Empty bins will be represented by NaN.
- function : a user-defined function which takes a 1D array of values, and outputs a single numerical statistic. This function will be called on the values in each bin. Empty bins will be represented by function([]), or NaN if this returns an error.

bins : int or sequence of scalars, optional

If *bins* is an int, it defines the number of equal-width bins in the given range (10 by default). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths. Values in *x* that are smaller than lowest bin edge are assigned to bin number 0, values beyond the highest bin are assigned to `bins[-1]`. If the bin edges are specified, the number of bins will be, (nx = len(bins)-1).

range : (float, float) or [(float, float)], optional

The lower and upper range of the bins. If not provided, range is simply `(x.min(), x.max())`. Values outside the range are ignored.

Returns

statistic : array

The values of the selected statistic in each bin.

bin_edges : array of dtype float

Return the bin edges `(length(statistic)+1)`.

binnumber: 1-D ndarray of ints

Indices of the bins (corresponding to *bin_edges*) in which each value of *x* belongs. Same length as *values*. A binnumber of *i* means the corresponding value is between (bin_edges[i-1], bin_edges[i]).

See also:

`numpy.digitize`, `numpy.histogram`, *binned_statistic_2d*, *binned_statistic_dd*

Notes

All but the last (righthand-most) bin is half-open. In other words, if *bins* is `[1, 2, 3, 4]`, then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

New in version 0.11.0.

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

First some basic examples:

Create two evenly spaced bins in the range of the given sample, and sum the corresponding values in each of those bins:

```
>>> values = [1.0, 1.0, 2.0, 1.5, 3.0]
>>> stats.binned_statistic([1, 1, 2, 5, 7], values, 'sum', bins=2)
(array([ 4. ,  4.5]), array([ 1.,  4.,  7.]), array([1, 1, 1, 2, 2]))
```

Multiple arrays of values can also be passed. The statistic is calculated on each set independently:

```
>>> values = [[1.0, 1.0, 2.0, 1.5, 3.0], [2.0, 2.0, 4.0, 3.0, 6.0]]
>>> stats.binned_statistic([1, 1, 2, 5, 7], values, 'sum', bins=2)
```

```
(array([[ 4. ,  4.5], [ 8. ,  9. ]]), array([ 1.,  4.,  7.]),
    array([1, 1, 1, 2, 2]))
```

```
>>> stats.binned_statistic([1, 2, 1, 2, 4], np.arange(5), statistic='mean',
...                        bins=3)
(array([ 1.,  2.,  4.]), array([ 1.,  2.,  3.,  4.]),
    array([1, 2, 1, 2, 3]))
```

As a second example, we now generate some random data of sailing boat speed as a function of wind speed, and then determine how fast our boat is for certain wind speeds:

```
>>> windspeed = 8 * np.random.rand(500)
>>> boatspeed = .3 * windspeed**.5 + .2 * np.random.rand(500)
>>> bin_means, bin_edges, binnumber = stats.binned_statistic(windspeed,
...                 boatspeed, statistic='median', bins=[1,2,3,4,5,6,7])
>>> plt.figure()
>>> plt.plot(windspeed, boatspeed, 'b.', label='raw data')
>>> plt.hlines(bin_means, bin_edges[:-1], bin_edges[1:], colors='g', lw=5,
...            label='binned statistic of data')
>>> plt.legend()
```

Now we can use binnumber to select all datapoints with a windspeed below 1:

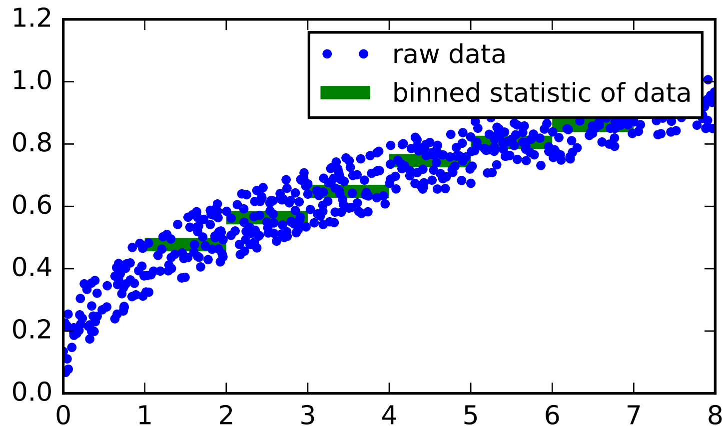
```
>>> low_boatspeed = boatspeed[binnumber == 0]
```

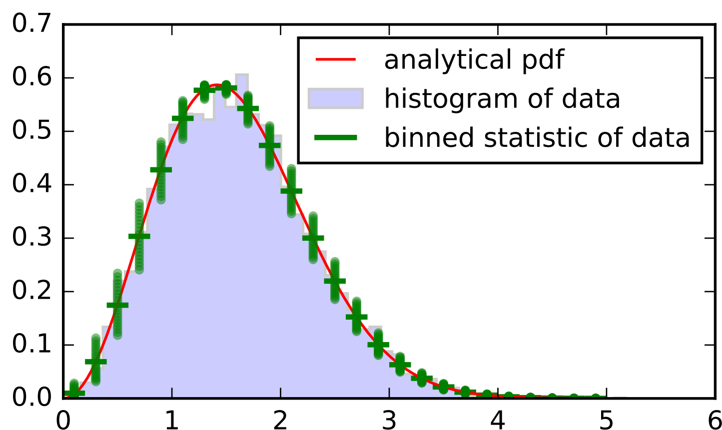
As a final example, we will use bin_edges and binnumber to make a plot of a distribution that shows the mean and distribution around that mean per bin, on top of a regular histogram and the probability distribution function:

```
>>> x = np.linspace(0, 5, num=500)
>>> x_pdf = stats.maxwell.pdf(x)
>>> samples = stats.maxwell.rvs(size=10000)
```

```
>>> bin_means, bin_edges, binnumber = stats.binned_statistic(x, x_pdf,
...         statistic='mean', bins=25)
>>> bin_width = (bin_edges[1] - bin_edges[0])
>>> bin_centers = bin_edges[1:] - bin_width/2
```

```
>>> plt.figure()
>>> plt.hist(samples, bins=50, normed=True, histtype='stepfilled',
...          alpha=0.2, label='histogram of data')
>>> plt.plot(x, x_pdf, 'r-', label='analytical pdf')
>>> plt.hlines(bin_means, bin_edges[:-1], bin_edges[1:], colors='g', lw=2,
...            label='binned statistic of data')
>>> plt.plot((binnumber - 0.5) * bin_width, x_pdf, 'g.', alpha=0.5)
>>> plt.legend(fontsize=10)
>>> plt.show()
```





`scipy.stats.`**`binned_statistic_2d`**(*x*, *y*, *values*, *statistic='mean'*, *bins=10*, *range=None*, *expand_binnumbers=False*)

Compute a bidimensional binned statistic for one or more sets of data.

This is a generalization of a histogram2d function. A histogram divides the space into bins, and returns the count of the number of points in each bin. This function allows the computation of the sum, mean, median, or other statistic of the values (or set of values) within each bin.

Parameters

x : (N,) array_like
A sequence of values to be binned along the first dimension.

y : (N,) array_like
A sequence of values to be binned along the second dimension.

values : (N,) array_like or list of (N,) array_like
The data on which the statistic will be computed. This must be the same shape as *x*, or a list of sequences - each with the same shape as *x*. If *values* is such a list, the statistic will be computed on each independently.

statistic : string or callable, optional

The statistic to compute (default is 'mean'). The following statistics are available:

- 'mean' : compute the mean of values for points within each bin. Empty bins will be represented by NaN.
- 'median' : compute the median of values for points within each bin. Empty bins will be represented by NaN.
- 'count' : compute the count of points within each bin. This is identical to an unweighted histogram. *values* array is not referenced.
- 'sum' : compute the sum of values for points within each bin. This is identical to a weighted histogram.
- 'min' : compute the minimum of values for points within each bin. Empty bins will be represented by NaN..
- 'max' : compute the maximum of values for point within each bin. Empty bins will be represented by NaN.
- function : a user-defined function which takes a 1D array of values, and outputs a single numerical statistic. This function will be called on the values in each bin. Empty bins will be represented by function([]), or NaN if this returns an error.

bins : int or [int, int] or array_like or [array, array], optional

The bin specification:

- the number of bins for the two dimensions (nx = ny = bins),
- the number of bins in each dimension (nx, ny = bins),
- the bin edges for the two dimensions (x_edge = y_edge = bins),
- the bin edges in each dimension (x_edge, y_edge = bins).

If the bin edges are specified, the number of bins will be, (nx = len(x_edge)-1, ny = len(y_edge)-1).

range : (2,2) array_like, optional

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the *bins* parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

expand_binnumbers : bool, optional

'False' (default): the returned *binnumber* is a shape (N,) array of linearized bin indices. 'True': the returned *binnumber* is 'unraveled' into a shape (2,N) ndarray, where each row gives the bin numbers in the corresponding dimension. See the *binnumber* returned value, and the *Examples* section.

New in version 0.17.0.

Returns

statistic : (nx, ny) ndarray

The values of the selected statistic in each two-dimensional bin.

x_edge : (nx + 1) ndarray

The bin edges along the first dimension.

y_edge : (ny + 1) ndarray

The bin edges along the second dimension.

binnumber : (N,) array of ints or (2,N) ndarray of ints

This assigns to each element of *sample* an integer that represents the bin in which this observation falls. The representation depends on the *expand_binnumbers* argument. See *Notes* for details.

See also:

`numpy.digitize`, `numpy.histogram2d`, *binned_statistic*, *binned_statistic_dd*

Notes

Binedges: All but the last (righthand-most) bin is half-open. In other words, if *bins* is `[1, 2, 3, 4]`, then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

binnumber: This returned argument assigns to each element of *sample* an integer that represents the bin in which it belongs. The representation depends on the *expand_binnumbers* argument. If 'False' (default): The returned *binnumber* is a shape (N,) array of linearized indices mapping each element of *sample* to its corresponding bin (using row-major ordering). If 'True': The returned *binnumber* is a shape (2,N) ndarray where each row indicates bin placements for each dimension respectively. In each dimension, a binnumber of *i* means the corresponding value is between (D_edge[i-1], D_edge[i]), where 'D' is either 'x' or 'y'.

New in version 0.11.0.

Examples

```
>>> from scipy import stats
```

Calculate the counts with explicit bin-edges:

```
>>> x = [0.1, 0.1, 0.1, 0.6]
>>> y = [2.1, 2.6, 2.1, 2.1]
>>> binx = [0.0, 0.5, 1.0]
>>> biny = [2.0, 2.5, 3.0]
>>> ret = stats.binned_statistic_2d(x, y, None, 'count', bins=[binx,biny])
>>> ret.statistic
array([[ 2.,  1.],
       [ 1.,  0.]])
```

The bin in which each sample is placed is given by the *binnumber* returned parameter. By default, these are the linearized bin indices:

```
>>> ret.binnumber
array([5, 6, 5, 9])
```

The bin indices can also be expanded into separate entries for each dimension using the *expand_binnumbers* parameter:

```
>>> ret = stats.binned_statistic_2d(x, y, None, 'count', bins=[binx,biny],
...                                 expand_binnumbers=True)
>>> ret.binnumber
array([[1, 1, 1, 2],
       [1, 2, 1, 1]])
```

Which shows that the first three elements belong in the xbin 1, and the fourth into xbin 2; and so on for y.

`scipy.stats.`**`binned_statistic_dd`**(*sample*, *values*, *statistic='mean'*, *bins=10*, *range=None*, *expand_binnumbers=False*)

Compute a multidimensional binned statistic for a set of data.

This is a generalization of a histogramdd function. A histogram divides the space into bins, and returns the count of the number of points in each bin. This function allows the computation of the sum, mean, median, or other statistic of the values within each bin.

Parameters

sample : array_like
: Data to histogram passed as a sequence of D arrays of length N, or as an (N,D) array.

values : (N,) array_like or list of (N,) array_like
: The data on which the statistic will be computed. This must be the same shape as *x*, or a list of sequences - each with the same shape as *x*. If *values* is such a list, the statistic will be computed on each independently.

statistic : string or callable, optional
: The statistic to compute (default is 'mean'). The following statistics are available:

- 'mean' : compute the mean of values for points within each bin. Empty bins will be represented by NaN.
- 'median' : compute the median of values for points within each bin. Empty bins will be represented by NaN.
- 'count' : compute the count of points within each bin. This is identical to an unweighted histogram. *values* array is not referenced.
- 'sum' : compute the sum of values for points within each bin. This is identical to a weighted histogram.
- 'min' : compute the minimum of values for points within each bin. Empty bins will be represented by NaN..
- 'max' : compute the maximum of values for point within each bin. Empty bins will be represented by NaN.
- function : a user-defined function which takes a 1D array of values, and outputs a single numerical statistic. This function will be called on the values in each bin. Empty bins will be represented by function([]), or NaN if this returns an error.

bins : sequence or int, optional

The bin specification must be in one of the following forms:

- A sequence of arrays describing the bin edges along each dimension.
- The number of bins for each dimension (nx, ny, ... = bins).
- The number of bins for all dimensions (nx = ny = ... = bins).

range : sequence, optional

A sequence of lower and upper bin edges to be used if the edges are not given explicitely in *bins*. Defaults to the minimum and maximum values along each dimension.

expand_binnumbers : bool, optional

'False' (default): the returned *binnumber* is a shape (N,) array of linearized bin indices. 'True': the returned *binnumber* is 'unraveled' into a shape (D,N) ndarray, where each row gives the bin numbers in the corresponding dimension. See the *binnumber* returned value, and the *Examples* section of `binned_statistic_2d`.

New in version 0.17.0.

Returns

statistic : ndarray, shape(nx1, nx2, nx3,...)

The values of the selected statistic in each two-dimensional bin.

bin_edges : list of ndarrays

A list of D arrays describing the (nxi + 1) bin edges for each dimension.

binnumber : (N,) array of ints or (D,N) ndarray of ints

This assigns to each element of *sample* an integer that represents the bin in which this observation falls. The representation depends on the *expand_binnumbers* argument. See *Notes* for details.

See also:

`numpy.digitize`, `numpy.histogramdd`, `binned_statistic`, `binned_statistic_2d`

Notes

Binedges: All but the last (righthand-most) bin is half-open in each dimension. In other words, if *bins* is `[1, 2, 3, 4]`, then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

binnumber: This returned argument assigns to each element of *sample* an integer that represents the bin in which it belongs. The representation depends on the *expand_binnumbers* argument. If 'False' (default): The returned *binnumber* is a shape (N,) array of linearized indices mapping each element of *sample* to its corresponding bin (using row-major ordering). If 'True': The returned *binnumber* is a shape (D,N) ndarray where each row indicates bin placements for each dimension respectively. In each dimension, a binnumber of *i* means the corresponding value is between (bin_edges[D][i-1], bin_edges[D][i]), for each dimension 'D'.

New in version 0.11.0.

obrientransform(*args)	Computes the O'Brien transform on input data (any number of arrays).
signaltonoise(*args, **kwds)	*signaltonoise* is deprecated!
bayes_mvs(data[, alpha])	Bayesian confidence intervals for the mean, var, and std.
mvsdist(data)	'Frozen' distributions for mean, variance, and standard deviation of data.
sem(a[, axis, ddof, nan_policy])	Calculates the standard error of the mean (or standard error of measurement) of the values in the input array.
zmap(scores, compare[, axis, ddof])	Calculates the relative z-scores.
zscore(a[, axis, ddof])	Calculates the z score of each value in the sample, relative to the sample mean and standard deviation.
iqr(x[, axis, rng, scale, nan_policy, ...])	Compute the interquartile range of the data along the specified axis.

scipy.stats.**obrientransform**(**args*)

Computes the O'Brien transform on input data (any number of arrays).

Used to test for homogeneity of variance prior to running one-way stats. Each array in `*args` is one level of a factor. If *f_oneway* is run on the transformed data and found significant, the variances are unequal. From Maxwell and Delaney *[R616]*, p.112.

Parameters **args** : tuple of array_like

Any number of arrays.

Returns **obrientransform** : ndarray

Transformed data for use in an ANOVA. The first dimension of the result corresponds to the sequence of transformed arrays. If the arrays given are all 1-D of the same length, the return value is a 2-D array; otherwise it is a 1-D array of type object, with each element being an ndarray.

References

[R616]

Examples

We'll test the following data sets for differences in their variance.

```
>>> x = [10, 11, 13, 9, 7, 12, 12, 9, 10]
>>> y = [13, 21, 5, 10, 8, 14, 10, 12, 7, 15]
```

Apply the O'Brien transform to the data.

```
>>> from scipy.stats import obrientransform
>>> tx, ty = obrientransform(x, y)
```

Use *scipy.stats.f_oneway* to apply a one-way ANOVA test to the transformed data.

```
>>> from scipy.stats import f_oneway
>>> F, p = f_oneway(tx, ty)
>>> p
0.1314139477040335
```

If we require that `p < 0.05` for significance, we cannot conclude that the variances are different.

scipy.stats.**signaltonoise**(**args*, ***kwds*)

signaltonoise is deprecated! scipy.stats.signaltonoise is deprecated in scipy 0.16.0

The signal-to-noise ratio of the input data.

Returns the signal-to-noise ratio of *a*, here defined as the mean divided by the standard deviation.

Parameters
- **a** : array_like
  An array_like object containing the sample data.
- ***axis*** : [int or None, optional] Axis along which to operate. Default is 0. If None, compute over the whole array *a*.
- ***ddof*** : [int, optional] Degrees of freedom correction for standard deviation. Default is 0.

Returns
- **s2n** : ndarray
  The mean to standard deviation ratio(s) along *axis*, or 0 where the standard deviation is 0.

`scipy.stats.bayes_mvs(data, alpha=0.9)`

Bayesian confidence intervals for the mean, var, and std.

Parameters
- **data** : array_like
  Input data, if multi-dimensional it is flattened to 1-D by `bayes_mvs`. Requires 2 or more data points.
- **alpha** : float, optional
  Probability that the returned confidence interval contains the true parameter.

Returns
- **mean_cntr, var_cntr, std_cntr** : tuple
  The three results are for the mean, variance and standard deviation, respectively. Each result is a tuple of the form:

```
(center, (lower, upper))
```

  with *center* the mean of the conditional pdf of the value given the data, and *(lower, upper)* a confidence interval, centered on the median, containing the estimate to a probability `alpha`.

See also:

`mvsdist`

Notes

Each tuple of mean, variance, and standard deviation estimates represent the (center, (lower, upper)) with center the mean of the conditional pdf of the value given the data and (lower, upper) is a confidence interval centered on the median, containing the estimate to a probability `alpha`.

Converts data to 1-D and assumes all data has the same mean and variance. Uses Jeffrey's prior for variance and std.

Equivalent to `tuple((x.mean(), x.interval(alpha)) for x in mvsdist(dat))`

References

T.E. Oliphant, "A Bayesian perspective on estimating mean, variance, and standard-deviation from data", http://scholarsarchive.byu.edu/facpub/278, 2006.

Examples

First a basic example to demonstrate the outputs:

```
>>> from scipy import stats
>>> data = [6, 9, 12, 7, 8, 8, 13]
>>> mean, var, std = stats.bayes_mvs(data)
>>> mean
Mean(statistic=9.0, minmax=(7.1036502226125329, 10.896349777387467))
```

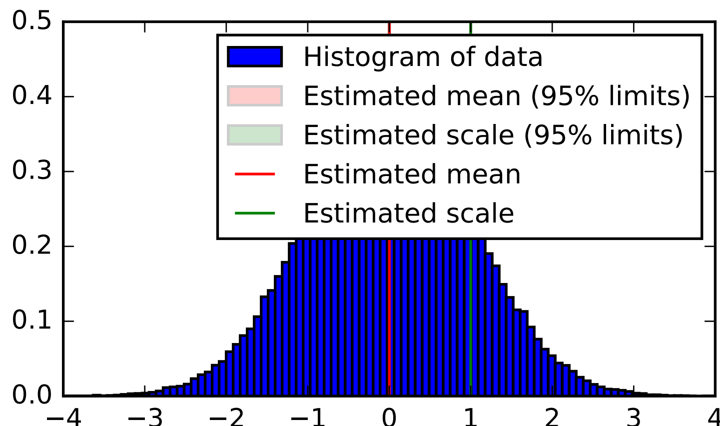
```
>>> var
Variance(statistic=10.0, minmax=(3.176724206..., 24.45910382...))
>>> std
Std_dev(statistic=2.9724954732045084, minmax=(1.7823367265645143, 4.
↪9456146050146295))
```

Now we generate some normally distributed random data, and get estimates of mean and standard deviation with 95% confidence intervals for those estimates:

```
>>> n_samples = 100000
>>> data = stats.norm.rvs(size=n_samples)
>>> res_mean, res_var, res_std = stats.bayes_mvs(data, alpha=0.95)
```

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.hist(data, bins=100, normed=True, label='Histogram of data')
>>> ax.vlines(res_mean.statistic, 0, 0.5, colors='r', label='Estimated mean')
>>> ax.axvspan(res_mean.minmax[0],res_mean.minmax[1], facecolor='r',
...            alpha=0.2, label=r'Estimated mean (95% limits)')
>>> ax.vlines(res_std.statistic, 0, 0.5, colors='g', label='Estimated scale')
>>> ax.axvspan(res_std.minmax[0],res_std.minmax[1], facecolor='g', alpha=0.2,
...            label=r'Estimated scale (95% limits)')
```

```
>>> ax.legend(fontsize=10)
>>> ax.set_xlim([-4, 4])
>>> ax.set_ylim([0, 0.5])
>>> plt.show()
```



scipy.stats.**mvsdist**(*data*)

‘Frozen’ distributions for mean, variance, and standard deviation of data.

Parameters **data** : array_like

Input array. Converted to 1-D using ravel. Requires 2 or more data-points.

Returns **mdist** : “frozen” distribution object

Distribution object representing the mean of the data

vdist : “frozen” distribution object

Distribution object representing the variance of the data

sdist : "frozen" distribution object

Distribution object representing the standard deviation of the data

See also:

bayes_mvs

Notes

The return values from `bayes_mvs(data)` is equivalent to `tuple((x.mean(), x.interval(0.90)) for x in mvsdist(data))`.

In other words, calling `<dist>.mean()` and `<dist>.interval(0.90)` on the three distribution objects returned from this function will give the same results that are returned from *bayes_mvs*.

References

T.E. Oliphant, "A Bayesian perspective on estimating mean, variance, and standard-deviation from data", http://scholarsarchive.byu.edu/facpub/278, 2006.

Examples

```
>>> from scipy import stats
>>> data = [6, 9, 12, 7, 8, 8, 13]
>>> mean, var, std = stats.mvsdist(data)
```

We now have frozen distribution objects "mean", "var" and "std" that we can examine:

```
>>> mean.mean()
9.0
>>> mean.interval(0.95)
(6.6120585482655692, 11.387941451734431)
>>> mean.std()
1.1952286093343936
```

`scipy.stats.sem(a, axis=0, ddof=1, nan_policy='propagate')`

Calculates the standard error of the mean (or standard error of measurement) of the values in the input array.

Parameters

a : array_like

An array containing the values for which the standard error is returned.

axis : int or None, optional

Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional

Delta degrees-of-freedom. How many degrees of freedom to adjust for bias in limited samples relative to the population estimate of variance. Defaults to 1.

nan_policy : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

s : ndarray or float

The standard error of the mean in the sample(s), along the input axis.

Notes

The default value for *ddof* is different to the default (0) used by other ddof containing routines, such as np.std and np.nanstd.

Examples

Find standard error along the first axis:

```
>>> from scipy import stats
>>> a = np.arange(20).reshape(5,4)
>>> stats.sem(a)
array([ 2.8284,  2.8284,  2.8284,  2.8284])
```

Find standard error across the whole array, using n degrees of freedom:

```
>>> stats.sem(a, axis=None, ddof=0)
1.2893796958227628
```

`scipy.stats.zmap(scores, compare, axis=0, ddof=0)`

Calculates the relative z-scores.

Returns an array of z-scores, i.e., scores that are standardized to zero mean and unit variance, where mean and variance are calculated from the comparison array.

Parameters

scores : array_like
: The input for which z-scores are calculated.

compare : array_like
: The input from which the mean and standard deviation of the normalization are taken; assumed to have the same dimension as *scores*.

axis : int or None, optional
: Axis over which mean and variance of *compare* are calculated. Default is 0. If None, compute over the whole array *scores*.

ddof : int, optional
: Degrees of freedom correction in the calculation of the standard deviation. Default is 0.

Returns

zscore : array_like
: Z-scores, in the same shape as *scores*.

Notes

This function preserves ndarray subclasses, and works also with matrices and masked arrays (it uses *asanyarray* instead of *asarray* for parameters).

Examples

```
>>> from scipy.stats import zmap
>>> a = [0.5, 2.0, 2.5, 3]
>>> b = [0, 1, 2, 3, 4]
>>> zmap(a, b)
array([-1.06066017,  0.        ,  0.35355339,  0.70710678])
```

`scipy.stats.zscore(a, axis=0, ddof=0)`

Calculates the z score of each value in the sample, relative to the sample mean and standard deviation.

Parameters

a : array_like
: An array like object containing the sample data.

axis : int or None, optional
: Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
: Degrees of freedom correction in the calculation of the standard deviation. Default is 0.

Returns

zscore : array_like

The z-scores, standardized by mean and standard deviation of input array *a*.

Notes

This function preserves ndarray subclasses, and works also with matrices and masked arrays (it uses *asanyarray* instead of *asarray* for parameters).

Examples

```
>>> a = np.array([ 0.7972,  0.0767,  0.4383,  0.7866,  0.8091,
...                0.1954,  0.6307,  0.6599,  0.1065,  0.0508])
>>> from scipy import stats
>>> stats.zscore(a)
array([ 1.1273, -1.247 , -0.0552,  1.0923,  1.1664, -0.8559,  0.5786,
        0.6748, -1.1488, -1.3324])
```

Computing along a specified axis, using n-1 degrees of freedom (`ddof=1`) to calculate the standard deviation:

```
>>> b = np.array([[ 0.3148,  0.0478,  0.6243,  0.4608],
...               [ 0.7149,  0.0775,  0.6072,  0.9656],
...               [ 0.6341,  0.1403,  0.9759,  0.4064],
...               [ 0.5918,  0.6948,  0.904 ,  0.3721],
...               [ 0.0921,  0.2481,  0.1188,  0.1366]])
>>> stats.zscore(b, axis=1, ddof=1)
array([[-0.19264823, -1.28415119,  1.07259584,  0.40420358],
       [ 0.33048416, -1.37380874,  0.04251374,  1.00081084],
       [ 0.26796377, -1.12598418,  1.23283094, -0.37481053],
       [-0.22095197,  0.24468594,  1.19042819, -1.21416216],
       [-0.82780366,  1.4457416 , -0.43867764, -0.1792603 ]])
```

`scipy.stats.`**`iqr`**(*x*, *axis=None*, *rng=(25, 75)*, *scale='raw'*, *nan_policy='propagate'*, *interpolation='linear'*, *keepdims=False*)

Compute the interquartile range of the data along the specified axis.

The interquartile range (IQR) is the difference between the 75th and 25th percentile of the data. It is a measure of the dispersion similar to standard deviation or variance, but is much more robust against outliers *[R592]*.

The `rng` parameter allows this function to compute other percentile ranges than the actual IQR. For example, setting `rng=(0, 100)` is equivalent to `numpy.ptp`.

The IQR of an empty array is *np.nan*.

New in version 0.18.0.

Parameters

x : array_like
: Input array or object that can be converted to an array.

axis : int or sequence of int, optional
: Axis along which the range is computed. The default is to compute the IQR for the entire array.

rng : Two-element sequence containing floats in range of [0,100] optional
: Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The default is the true IQR: *(25, 75)*. The order of the elements is not important.

scale : scalar or str, optional
: The numerical value of scale will be divided out of the final result. The following string values are recognized:

  'raw' : No scaling, just return the raw IQR. 'normal' : Scale by $2\sqrt{2}erf^{-1}(\frac{1}{2}) \approx 1.349$.

The default is 'raw'. Array-like scale is also allowed, as long as it broadcasts correctly to the output such that `out / scale` is a valid operation. The output dimensions depend on the input array, *x*, the *axis* argument, and the *keepdims* flag.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}, optional
Specifies the interpolation method to use when the percentile boundaries lie between two data points *i* and *j*:

- ***'linear'*** [*i* + *(j - i)* * *fraction*, where *fraction* is the] fractional part of the index surrounded by *i* and *j*.
- 'lower' : *i*.
- 'higher' : *j*.
- 'nearest' : *i* or *j* whichever is nearest.
- 'midpoint' : *(i + j) / 2*.

Default is 'linear'.

keepdims : bool, optional
If this is set to *True*, the reduced axes are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array *x*.

Returns **iqr** : scalar or ndarray
If `axis=None`, a scalar is returned. If the input contains integers or floats of smaller precision than `np.float64`, then the output data-type is `np.float64`. Otherwise, the output data-type is the same as that of the input.

See also:

`numpy.std`, `numpy.var`

Notes

This function is heavily dependent on the version of `numpy` that is installed. Versions greater than 1.11.0b3 are highly recommended, as they include a number of enhancements and fixes to `numpy.percentile` and `numpy.nanpercentile` that affect the operation of this function. The following modifications apply:

Below 1.10.0 [*nan_policy* is poorly defined.] The default behavior of `numpy.percentile` is used for 'propagate'. This is a hybrid of 'omit' and 'propagate' that mostly yields a skewed version of 'omit' since NaNs are sorted to the end of the data. A warning is raised if there are NaNs in the data.

Below 1.9.0: `numpy.nanpercentile` does not exist.
This means that `numpy.percentile` is used regardless of *nan_policy* and a warning is issued. See previous item for a description of the behavior.

Below 1.9.0:* keepdims *and* interpolation *are not supported.
The keywords get ignored with a warning if supplied with non-default values. However, multiple axes are still supported.

References

[R591], *[R592]*, *[R593]*

Examples

```
>>> from scipy.stats import iqr
>>> x = np.array([[10, 7, 4], [3, 2, 1]])
>>> x
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> iqr(x)
```

```
4.0
>>> iqr(x, axis=0)
array([ 3.5,  2.5,  1.5])
>>> iqr(x, axis=1)
array([ 3.,  1.])
>>> iqr(x, axis=1, keepdims=True)
array([[ 3.],
       [ 1.]])
```

`sigmaclip`(a[, low, high])	Iterative sigma-clipping of array elements.
`threshold`(*args, **kwds)	`threshold` is deprecated!
`trimboth`(a, proportiontocut[, axis])	Slices off a proportion of items from both ends of an array.
`trim1`(a, proportiontocut[, tail, axis])	Slices off a proportion from ONE end of the passed array distribution.

`scipy.stats.`**`sigmaclip`**(*a*, *low=4.0*, *high=4.0*)

Iterative sigma-clipping of array elements.

The output array contains only those elements of the input array *c* that satisfy the conditions

```
mean(c) - std(c)*low < c < mean(c) + std(c)*high
```

Starting from the full sample, all elements outside the critical range are removed. The iteration continues with a new critical range until no elements are outside the range.

Parameters

a : array_like

Data array, will be raveled if not 1-D.

low : float, optional

Lower bound factor of sigma clipping. Default is 4.

high : float, optional

Upper bound factor of sigma clipping. Default is 4.

Returns

clipped : ndarray

Input array with clipped elements removed.

lower : float

Lower threshold value use for clipping.

upper : float

Upper threshold value use for clipping.

Examples

```
>>> from scipy.stats import sigmaclip
>>> a = np.concatenate((np.linspace(9.5, 10.5, 31),
...                     np.linspace(0, 20, 5)))
>>> fact = 1.5
>>> c, low, upp = sigmaclip(a, fact, fact)
>>> c
array([  9.96666667,  10.        ,  10.03333333,  10.        ])
>>> c.var(), c.std()
(0.00055555555555555165, 0.023570226039551501)
>>> low, c.mean() - fact*c.std(), c.min()
(9.9646446609406727, 9.9646446609406727, 9.9666666666666668)
>>> upp, c.mean() + fact*c.std(), c.max()
(10.035355339059327, 10.035355339059327, 10.033333333333333)
```

```
>>> a = np.concatenate((np.linspace(9.5, 10.5, 11),
...                     np.linspace(-100, -50, 3)))
```

```
>>> c, low, upp = sigmaclip(a, 1.8, 1.8)
>>> (c == np.linspace(9.5, 10.5, 11)).all()
True
```

`scipy.stats.threshold(*args, **kwds)`

threshold is deprecated! stats.threshold is deprecated in scipy 0.17.0

Clip array to a given value.

Similar to numpy.clip(), except that values less than *threshmin* or greater than *threshmax* are replaced by *newval*, instead of by *threshmin* and *threshmax* respectively.

Parameters

a : array_like

Data to threshold.

threshmin [float, int or None, optional] Minimum threshold, defaults to None.

threshmax [float, int or None, optional] Maximum threshold, defaults to None.

newval [float or int, optional] Value to put in place of values in *a* outside of bounds. Defaults to 0.

Returns

out : ndarray

The clipped input array, with values less than *threshmin* or greater than *threshmax* replaced with *newval*.

Examples

```
>>> a = np.array([9, 9, 6, 3, 1, 6, 1, 0, 0, 8])
>>> from scipy import stats
>>> stats.threshold(a, threshmin=2, threshmax=8, newval=-1)
array([-1, -1,  6,  3, -1,  6, -1, -1, -1,  8])
```

`scipy.stats.trimboth(a, proportiontocut, axis=0)`

Slices off a proportion of items from both ends of an array.

Slices off the passed proportion of items from both ends of the passed array (i.e., with *proportiontocut* = 0.1, slices leftmost 10% **and** rightmost 10% of scores). The trimmed values are the lowest and highest ones. Slices off less if proportion results in a non-integer slice index (i.e., conservatively slices off'proportiontocut').

Parameters

a : array_like

Data to trim.

proportiontocut : float

Proportion (in range 0-1) of total data set to trim of each end.

axis : int or None, optional

Axis along which to trim data. Default is 0. If None, compute over the whole array *a*.

Returns

out : ndarray

Trimmed version of array *a*. The order of the trimmed content is undefined.

See also:

trim_mean

Examples

```
>>> from scipy import stats
>>> a = np.arange(20)
>>> b = stats.trimboth(a, 0.1)
>>> b.shape
(16,)
```

`scipy.stats.`**`trim1`**(*a*, *proportiontocut*, *tail='right'*, *axis=0*)

Slices off a proportion from ONE end of the passed array distribution.

If *proportiontocut* = 0.1, slices off 'leftmost' or 'rightmost' 10% of scores. The lowest or highest values are trimmed (depending on the tail). Slices off less if proportion results in a non-integer slice index (i.e., conservatively slices off *proportiontocut*).

Parameters

a : array_like
  Input array

proportiontocut : float
  Fraction to cut off of 'left' or 'right' of distribution

tail : {'left', 'right'}, optional
  Defaults to 'right'.

axis : int or None, optional
  Axis along which to trim data. Default is 0. If None, compute over the whole array *a*.

Returns

trim1 : ndarray
  Trimmed version of array *a*. The order of the trimmed content is undefined.

f_oneway(*args)	Performs a 1-way ANOVA.
pearsonr(x, y)	Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.
spearmanr(a[, b, axis, nan_policy])	Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.
pointbiserialr(x, y)	Calculates a point biserial correlation coefficient and its p-value.
kendalltau(x, y[, initial_lexsort, nan_policy])	Calculates Kendall's tau, a correlation measure for ordinal data.
weightedtau(x, y[, rank, weigher, additive])	Computes a weighted version of Kendall's τ.
linregress(x[, y])	Calculate a linear least-squares regression for two sets of measurements.
theilslopes(y[, x, alpha])	Computes the Theil-Sen estimator for a set of points (x, y).
f_value(*args, **kwds)	*f_value* is deprecated!

`scipy.stats.`**`f_oneway`**(**args*)

Performs a 1-way ANOVA.

The one-way ANOVA tests the null hypothesis that two or more groups have the same population mean. The test is applied to samples from two or more groups, possibly with differing sizes.

Parameters

sample1, sample2, ... : array_like
  The sample measurements for each group.

Returns

statistic : float
  The computed F-value of the test.

pvalue : float
  The associated p-value from the F-distribution.

Notes

The ANOVA test has important assumptions that must be satisfied in order for the associated p-value to be valid.

1. The samples are independent.
2. Each sample is from a normally distributed population.
3. The population standard deviations of the groups are all equal. This property is known as homoscedasticity.

If these assumptions are not true for a given set of data, it may still be possible to use the Kruskal-Wallis H-test (`scipy.stats.kruskal`) although with some loss of power.

The algorithm is from Heiman[2], pp.394-7.

References

[R574], [R575], [R576]

Examples

```
>>> import scipy.stats as stats
```

[R576] Here are some data on a shell measurement (the length of the anterior adductor muscle scar, standardized by dividing by length) in the mussel Mytilus trossulus from five locations: Tillamook, Oregon; Newport, Oregon; Petersburg, Alaska; Magadan, Russia; and Tvarminne, Finland, taken from a much larger data set used in McDonald et al. (1991).

```
>>> tillamook = [0.0571, 0.0813, 0.0831, 0.0976, 0.0817, 0.0859, 0.0735,
...              0.0659, 0.0923, 0.0836]
>>> newport = [0.0873, 0.0662, 0.0672, 0.0819, 0.0749, 0.0649, 0.0835,
...            0.0725]
>>> petersburg = [0.0974, 0.1352, 0.0817, 0.1016, 0.0968, 0.1064, 0.105]
>>> magadan = [0.1033, 0.0915, 0.0781, 0.0685, 0.0677, 0.0697, 0.0764,
...            0.0689]
>>> tvarminne = [0.0703, 0.1026, 0.0956, 0.0973, 0.1039, 0.1045]
>>> stats.f_oneway(tillamook, newport, petersburg, magadan, tvarminne)
(7.1210194716424473, 0.00028122423145345439)
```

`scipy.stats.`**`pearsonr`**(*x*, *y*)

Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

The Pearson correlation coefficient measures the linear relationship between two datasets. Strictly speaking, Pearson's correlation requires that each dataset be normally distributed, and not necessarily zero-mean. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

x : (N,) array_like

Input

y : (N,) array_like

Input

Returns

r : float

Pearson's correlation coefficient

p-value : float

2-tailed p-value

References

http://www.statsoft.com/textbook/glosp.html#Pearson%20Correlation

`scipy.stats.`**`spearmanr`**(*a*, *b=None*, *axis=0*, *nan_policy='propagate'*)

Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.

The Spearman correlation is a nonparametric measure of the monotonicity of the relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Spearman correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters

a, b : 1D or 2D array_like, b is optional
: One or two 1-D or 2-D arrays containing multiple variables and observations. When these are 1-D, each represents a vector of observations of a single variable. For the behavior in the 2-D case, see under `axis`, below. Both arrays need to have the same length in the `axis` dimension.

axis : int or None, optional
: If axis=0 (default), then each column represents a variable, with observations in the rows. If axis=1, the relationship is transposed: each row represents a variable, while the columns contain observations. If axis=None, then both arrays will be raveled.

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

correlation : float or ndarray (2-D square)
: Spearman correlation matrix or correlation coefficient (if only 2 variables are given as parameters. Correlation matrix is square with length equal to total number of variables (columns or rows) in a and b combined.

pvalue : float
: The two-sided p-value for a hypothesis test whose null hypothesis is that two sets of data are uncorrelated, has same dimension as rho.

Notes

Changes in scipy 0.8.0: rewrite to add tie-handling, and axis.

References

[R638]

Examples

```
>>> from scipy import stats
>>> stats.spearmanr([1,2,3,4,5], [5,6,7,8,7])
(0.82078268166812329, 0.088587005313543798)
>>> np.random.seed(1234321)
>>> x2n = np.random.randn(100, 2)
>>> y2n = np.random.randn(100, 2)
>>> stats.spearmanr(x2n)
(0.059969996999699973, 0.55338590803773591)
>>> stats.spearmanr(x2n[:,0], x2n[:,1])
(0.059969996999699973, 0.55338590803773591)
>>> rho, pval = stats.spearmanr(x2n, y2n)
>>> rho
array([[ 1.        ,  0.05997   ,  0.18569457,  0.06258626],
       [ 0.05997   ,  1.        ,  0.110003  ,  0.02534653],
       [ 0.18569457,  0.110003  ,  1.        ,  0.03488749],
       [ 0.06258626,  0.02534653,  0.03488749,  1.        ]])
>>> pval
array([[ 0.        ,  0.55338591,  0.06435364,  0.53617935],
       [ 0.55338591,  0.        ,  0.27592895,  0.80234077],
       [ 0.06435364,  0.27592895,  0.        ,  0.73039992],
       [ 0.53617935,  0.80234077,  0.73039992,  0.        ]])
>>> rho, pval = stats.spearmanr(x2n.T, y2n.T, axis=1)
```

```
>>> rho
array([[ 1.        ,  0.05997   ,  0.18569457,  0.06258626],
       [ 0.05997   ,  1.        ,  0.110003  ,  0.02534653],
       [ 0.18569457,  0.110003  ,  1.        ,  0.03488749],
       [ 0.06258626,  0.02534653,  0.03488749,  1.        ]])
>>> stats.spearmanr(x2n, y2n, axis=None)
(0.10816770419260482, 0.1273562188027364)
>>> stats.spearmanr(x2n.ravel(), y2n.ravel())
(0.10816770419260482, 0.1273562188027364)
```

```
>>> xint = np.random.randint(10, size=(100, 2))
>>> stats.spearmanr(xint)
(0.052760927029710199, 0.60213045837062351)
```

`scipy.stats.`**`pointbiserialr`**(*x*, *y*)

Calculates a point biserial correlation coefficient and its p-value.

The point biserial correlation is used to measure the relationship between a binary variable, x, and a continuous variable, y. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply a determinative relationship.

This function uses a shortcut formula but produces the same result as *pearsonr*.

Parameters

x : array_like of bools
    Input array.

y : array_like
    Input array.

Returns

correlation : float
    R value

pvalue : float
    2-tailed p-value

Notes

pointbiserialr uses a t-test with `n-1` degrees of freedom. It is equivalent to *pearsonr.*

The value of the point-biserial correlation can be calculated from:

$$r_{pb} = \frac{\overline{Y_1} - \overline{Y_0}}{s_y}\sqrt{\frac{N_1 N_2}{N(N-1))}}$$

Where Y_0 and Y_1 are means of the metric observations coded 0 and 1 respectively; N_0 and N_1 are number of observations coded 0 and 1 respectively; N is the total number of observations and s_y is the standard deviation of all the metric observations.

A value of r_{pb} that is significantly different from zero is completely equivalent to a significant difference in means between the two groups. Thus, an independent groups t Test with $N-2$ degrees of freedom may be used to test whether r_{pb} is nonzero. The relation between the t-statistic for comparing two independent groups and r_{pb} is given by:

$$t = \sqrt{N-2}\frac{r_{pb}}{\sqrt{1-r^2_{pb}}}$$

References

[R618], [R619], [R620]

Examples

```
>>> from scipy import stats
>>> a = np.array([0, 0, 0, 1, 1, 1, 1])
>>> b = np.arange(7)
>>> stats.pointbiserialr(a, b)
(0.8660254037844386, 0.011724811003954652)
>>> stats.pearsonr(a, b)
(0.86602540378443871, 0.011724811003954626)
>>> np.corrcoef(a, b)
array([[ 1.       ,  0.8660254],
       [ 0.8660254,  1.       ]])
```

`scipy.stats.`**`kendalltau`**(*x*, *y*, *initial_lexsort=None*, *nan_policy='propagate'*)

Calculates Kendall's tau, a correlation measure for ordinal data.

Kendall's tau is a measure of the correspondence between two rankings. Values close to 1 indicate strong agreement, values close to -1 indicate strong disagreement. This is the 1945 "tau-b" version of Kendall's tau *[R596]*, which can account for ties and which reduces to the 1938 "tau-a" version *[R595]* in absence of ties.

Parameters

x, y : array_like
: Arrays of rankings, of the same shape. If arrays are not 1-D, they will be flattened to 1-D.

initial_lexsort : bool, optional
: Unused (deprecated).

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'. Note that if the input contains nan 'omit' delegates to mstats_basic.kendalltau(), which has a different implementation.

Returns

correlation : float
: The tau statistic.

pvalue : float
: The two-sided p-value for a hypothesis test whose null hypothesis is an absence of association, tau = 0.

See also:

`spearmanr` Calculates a Spearman rank-order correlation coefficient.

`theilslopes` Computes the Theil-Sen estimator for a set of points (x, y).

`weightedtau` Computes a weighted version of Kendall's tau.

Notes

The definition of Kendall's tau that is used is *[R596]*:

```
tau = (P - Q) / sqrt((P + Q + T) * (P + Q + U))
```

where P is the number of concordant pairs, Q the number of discordant pairs, T the number of ties only in *x*, and U the number of ties only in *y*. If a tie occurs for the same pair in both *x* and *y*, it is not added to either T or U.

References

[R595], *[R596]*, *[R597]*, *[R598]*

Examples

```
>>> from scipy import stats
>>> x1 = [12, 2, 1, 12, 2]
>>> x2 = [1, 4, 7, 1, 0]
>>> tau, p_value = stats.kendalltau(x1, x2)
>>> tau
-0.47140452079103173
>>> p_value
0.2827454599327748
```

`scipy.stats.`**`weightedtau`**(*x*, *y*, *rank=True*, *weigher=None*, *additive=True*)

Computes a weighted version of Kendall's τ.

The weighted τ is a weighted version of Kendall's τ in which exchanges of high weight are more influential than exchanges of low weight. The default parameters compute the additive hyperbolic version of the index, τ_h, which has been shown to provide the best balance between important and unimportant elements *[R648]*.

The weighting is defined by means of a rank array, which assigns a nonnegative rank to each element, and a weigher function, which assigns a weight based from the rank to each element. The weight of an exchange is then the sum or the product of the weights of the ranks of the exchanged elements. The default parameters compute τ_h: an exchange between elements with rank r and s (starting from zero) has weight $1/(r+1) + 1/(s+1)$.

Specifying a rank array is meaningful only if you have in mind an external criterion of importance. If, as it usually happens, you do not have in mind a specific rank, the weighted τ is defined by averaging the values obtained using the decreasing lexicographical rank by (*x*, *y*) and by (*y*, *x*). This is the behavior with default parameters.

Note that if you are computing the weighted τ on arrays of ranks, rather than of scores (i.e., a larger value implies a lower rank) you must negate the ranks, so that elements of higher rank are associated with a larger value.

Parameters

x, y : array_like

Arrays of scores, of the same shape. If arrays are not 1-D, they will be flattened to 1-D.

rank: array_like of ints or bool, optional

A nonnegative rank assigned to each element. If it is None, the decreasing lexicographical rank by (*x*, *y*) will be used: elements of higher rank will be those with larger *x*-values, using *y*-values to break ties (in particular, swapping *x* and *y* will give a different result). If it is False, the element indices will be used directly as ranks. The default is True, in which case this function returns the average of the values obtained using the decreasing lexicographical rank by (*x*, *y*) and by (*y*, *x*).

weigher : callable, optional

The weigher function. Must map nonnegative integers (zero representing the most important element) to a nonnegative weight. The default, None, provides hyperbolic weighing, that is, rank r is mapped to weight $1/(r+1)$.

additive : bool, optional

If True, the weight of an exchange is computed by adding the weights of the ranks of the exchanged elements; otherwise, the weights are multiplied. The default is True.

Returns

correlation : float

The weighted τ correlation index.

pvalue : float

Presently `np.nan`, as the null statistics is unknown (even in the additive hyperbolic case).

See also:

kendalltau Calculates Kendall's tau.
spearmanr Calculates a Spearman rank-order correlation coefficient.
theilslopes Computes the Theil-Sen estimator for a set of points (x, y).

Notes

This function uses an $O(n \log n)$, mergesort-based algorithm *[R648]* that is a weighted extension of Knight's algorithm for Kendall's τ *[R649]*. It can compute Shieh's weighted τ *[R650]* between rankings without ties (i.e., permutations) by setting *additive* and *rank* to False, as the definition given in *[R648]* is a generalization of Shieh's.

NaNs are considered the smallest possible score.

New in version 0.19.0.

References

[R648], *[R649]*, *[R650]*

Examples

```
>>> from scipy import stats
>>> x = [12, 2, 1, 12, 2]
>>> y = [1, 4, 7, 1, 0]
>>> tau, p_value = stats.weightedtau(x, y)
>>> tau
-0.56694968153682723
>>> p_value
nan
>>> tau, p_value = stats.weightedtau(x, y, additive=False)
>>> tau
-0.62205716951801038
```

NaNs are considered the smallest possible score:

```
>>> x = [12, 2, 1, 12, 2]
>>> y = [1, 4, 7, 1, np.nan]
>>> tau, _ = stats.weightedtau(x, y)
>>> tau
-0.56694968153682723
```

This is exactly Kendall's tau:

```
>>> x = [12, 2, 1, 12, 2]
>>> y = [1, 4, 7, 1, 0]
>>> tau, _ = stats.weightedtau(x, y, weigher=lambda x: 1)
>>> tau
-0.47140452079103173
```

```
>>> x = [12, 2, 1, 12, 2]
>>> y = [1, 4, 7, 1, 0]
>>> stats.weightedtau(x, y, rank=None)
WeightedTauResult(correlation=-0.4157652301037516, pvalue=nan)
>>> stats.weightedtau(y, x, rank=None)
WeightedTauResult(correlation=-0.71813413296990281, pvalue=nan)
```

scipy.stats.**linregress**(*x*, *y=None*)
Calculate a linear least-squares regression for two sets of measurements.

Parameters **x, y** : array_like

Two sets of measurements. Both arrays should have the same length. If only x is given (and y=None), then it must be a two-dimensional array where one dimension has length 2. The two sets of measurements are then found by splitting the array along the length-2 dimension.

Returns **slope** : float

slope of the regression line

intercept : float

intercept of the regression line

rvalue : float

correlation coefficient

pvalue : float

two-sided p-value for a hypothesis test whose null hypothesis is that the slope is zero.

stderr : float

Standard error of the estimated gradient.

See also:

scipy.optimize.curve_fit
Use non-linear least squares to fit a function to data.

scipy.optimize.leastsq
Minimize the sum of squares of a set of equations.

Examples

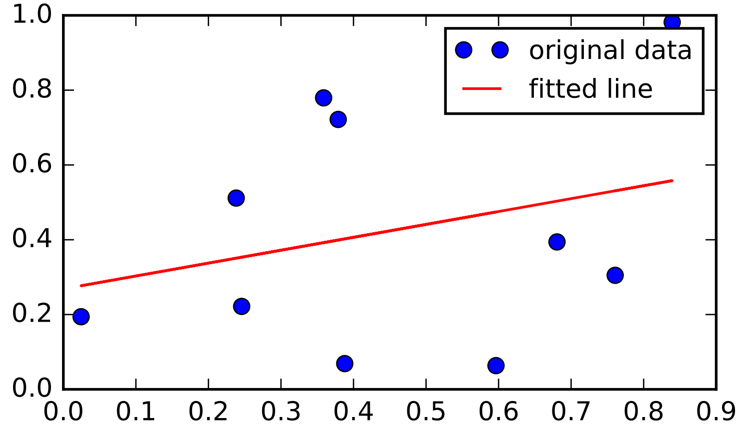
```
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> np.random.seed(12345678)
>>> x = np.random.random(10)
>>> y = np.random.random(10)
>>> slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
```

To get coefficient of determination (r_squared)

```
>>> print("r-squared:", r_value**2)
('r-squared:', 0.080402268539028335)
```

Plot the data along with the fitted line

```
>>> plt.plot(x, y, 'o', label='original data')
>>> plt.plot(x, intercept + slope*x, 'r', label='fitted line')
>>> plt.legend()
>>> plt.show()
```



`scipy.stats.theilslopes(y, x=None, alpha=0.95)`

Computes the Theil-Sen estimator for a set of points (x, y).

theilslopes implements a method for robust linear regression. It computes the slope as the median of all slopes between paired values.

Parameters

y : array_like
: Dependent variable.

x : array_like or None, optional
: Independent variable. If None, use `arange(len(y))` instead.

alpha : float, optional
: Confidence degree between 0 and 1. Default is 95% confidence. Note that *alpha* is symmetric around 0.5, i.e. both 0.1 and 0.9 are interpreted as "find the 90% confidence interval".

Returns

medslope : float
: Theil slope.

medintercept : float
: Intercept of the Theil line, as `median(y) - medslope*median(x)`.

lo_slope : float
: Lower bound of the confidence interval on *medslope*.

up_slope : float
: Upper bound of the confidence interval on *medslope*.

Notes

The implementation of *theilslopes* follows *[R639]*. The intercept is not defined in *[R639]*, and here it is defined as `median(y) - medslope*median(x)`, which is given in *[R641]*. Other definitions of the intercept exist in the literature. A confidence interval for the intercept is not given as this question is not addressed in *[R639]*.

References

[R639], *[R640]*, *[R641]*

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

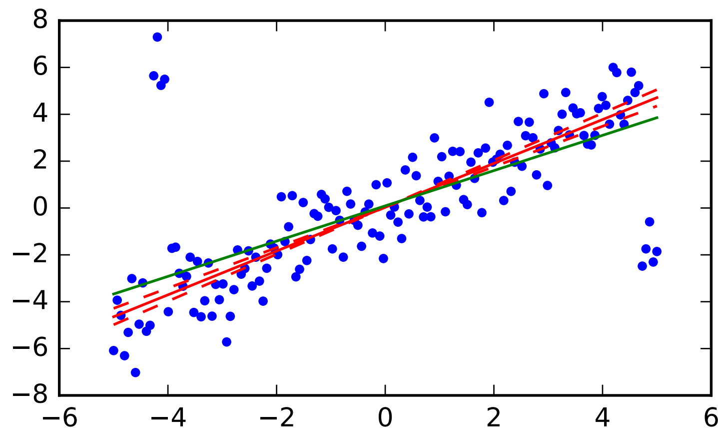
```
>>> x = np.linspace(-5, 5, num=150)
>>> y = x + np.random.normal(size=x.size)
>>> y[11:15] += 10  # add outliers
>>> y[-5:] -= 7
```

Compute the slope, intercept and 90% confidence interval. For comparison, also compute the least-squares fit with *linregress*:

```
>>> res = stats.theilslopes(y, x, 0.90)
>>> lsq_res = stats.linregress(x, y)
```

Plot the results. The Theil-Sen regression line is shown in red, with the dashed red lines illustrating the confidence interval of the slope (note that the dashed red lines are not the confidence interval of the regression as the confidence interval of the intercept is not included). The green line shows the least-squares fit for comparison.

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(x, y, 'b.')
>>> ax.plot(x, res[1] + res[0] * x, 'r-')
>>> ax.plot(x, res[1] + res[2] * x, 'r--')
>>> ax.plot(x, res[1] + res[3] * x, 'r--')
>>> ax.plot(x, lsq_res[1] + lsq_res[0] * x, 'g-')
>>> plt.show()
```



`scipy.stats.f_value(*args, **kwds)`

f_value is deprecated! stats.f_value deprecated in scipy 0.17.0

Returns an F-statistic for a restricted vs. unrestricted model.

Parameters **ER** : float

ER ***is the sum of squared residuals for the restricted model***

or null hypothesis

EF [float]

EF ***is the sum of squared residuals for the unrestricted model***

or alternate hypothesis

dfR [int] *dfR* is the degrees of freedom in the restricted model

dfF [int] *dfF* is the degrees of freedom in the unrestricted model

Returns **F-statistic** : float

ttest_1samp(a, popmean[, axis, nan_policy])	Calculates the T-test for the mean of ONE group of scores.
ttest_ind(a, b[, axis, equal_var, nan_policy])	Calculates the T-test for the means of *two independent* samples of scores.
ttest_ind_from_stats(mean1, std1, nobs1, ...)	T-test for means of two independent samples from descriptive statistics.
ttest_rel(a, b[, axis, nan_policy])	Calculates the T-test on TWO RELATED samples of scores, a and b.
kstest(rvs, cdf[, args, N, alternative, mode])	Perform the Kolmogorov-Smirnov test for goodness of fit.
chisquare(f_obs[, f_exp, ddof, axis])	Calculates a one-way chi square test.
power_divergence(f_obs[, f_exp, ddof, axis, ...])	Cressie-Read power divergence statistic and goodness of fit test.
ks_2samp(data1, data2)	Computes the Kolmogorov-Smirnov statistic on 2 samples.
mannwhitneyu(x, y[, use_continuity, alternative])	Computes the Mann-Whitney rank test on samples x and y.
tiecorrect(rankvals)	Tie correction factor for ties in the Mann-Whitney U and Kruskal-Wallis H tests.
rankdata(a[, method])	Assign ranks to data, dealing with ties appropriately.
ranksums(x, y)	Compute the Wilcoxon rank-sum statistic for two samples.
wilcoxon(x[, y, zero_method, correction])	Calculate the Wilcoxon signed-rank test.
kruskal(*args, **kwargs)	Compute the Kruskal-Wallis H-test for independent samples
friedmanchisquare(*args)	Computes the Friedman test for repeated measurements
combine_pvalues(pvalues[, method, weights])	Methods for combining the p-values of independent tests bearing upon the same hypothesis.
ss(*args, **kwds)	*ss* is deprecated!
square_of_sums(*args, **kwds)	*square_of_sums* is deprecated!
jarque_bera(x)	Perform the Jarque-Bera goodness of fit test on sample data.

scipy.stats.**ttest_1samp**(*a*, *popmean*, *axis=0*, *nan_policy='propagate'*)

Calculates the T-test for the mean of ONE group of scores.

This is a two-sided test for the null hypothesis that the expected value (mean) of a sample of independent observations *a* is equal to the given population mean, *popmean*.

Parameters

a : array_like

sample observation

popmean : float or array_like

expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int or None, optional

Axis along which to compute test. If None, compute over the whole array *a*.

nan_policy : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float or array

t-statistic

pvalue : float or array

two-tailed p-value

Examples

```
>>> from scipy import stats
```

```
>>> np.random.seed(7654567)  # fix seed to get the same result
>>> rvs = stats.norm.rvs(loc=5, scale=10, size=(50,2))
```

Test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case.

```
>>> stats.ttest_1samp(rvs,5.0)
(array([-0.68014479, -0.04323899]), array([ 0.49961383,  0.96568674]))
>>> stats.ttest_1samp(rvs,0.0)
(array([ 2.77025808,  4.11038784]), array([ 0.00789095,  0.00014999]))
```

Examples using axis and non-scalar dimension for population mean.

```
>>> stats.ttest_1samp(rvs,[5.0,0.0])
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs.T,[5.0,0.0],axis=1)
(array([-0.68014479,  4.11038784]), array([  4.99613833e-01,   1.49986458e-04]))
>>> stats.ttest_1samp(rvs,[[5.0],[0.0]])
(array([[-0.68014479, -0.04323899],
       [ 2.77025808,  4.11038784]]), array([[  4.99613833e-01,   9.65686743e-01],
       [  7.89094663e-03,   1.49986458e-04]]))
```

`scipy.stats.`**`ttest_ind`**(*a*, *b*, *axis=0*, *equal_var=True*, *nan_policy='propagate'*)

Calculates the T-test for the means of *two independent* samples of scores.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values. This test assumes that the populations have identical variances by default.

Parameters

a, b : array_like
: The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int or None, optional
: Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.

equal_var : bool, optional
: If True (default), perform a standard independent 2 sample test that assumes equal population variances *[R643]*. If False, perform Welch's t-test, which does not assume equal population variance *[R644]*.
: New in version 0.11.0.

nan_policy : {'propagate', 'raise', 'omit'}, optional
: Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float or array
: The calculated t-statistic.

pvalue : float or array
: The two-tailed p-value.

Notes

We can use this test, if we observe two independent samples from the same or different population, e.g. exam scores of boys and girls or of two ethnic groups. The test measures whether the average (expected) value differs significantly across samples. If we observe a large p-value, for example larger than 0.05 or 0.1, then we cannot

reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages.

References

[R643], [R644]

Examples

```
>>> from scipy import stats
>>> np.random.seed(12345678)
```

Test with sample with identical means:

```
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> rvs2 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> stats.ttest_ind(rvs1,rvs2)
(0.26833823296239279, 0.78849443369564776)
>>> stats.ttest_ind(rvs1,rvs2, equal_var = False)
(0.26833823296239279, 0.78849452749500748)
```

`ttest_ind` underestimates p for unequal variances:

```
>>> rvs3 = stats.norm.rvs(loc=5, scale=20, size=500)
>>> stats.ttest_ind(rvs1, rvs3)
(-0.46580283298287162, 0.64145827413436174)
>>> stats.ttest_ind(rvs1, rvs3, equal_var = False)
(-0.46580283298287162, 0.64149646246569292)
```

When n1 != n2, the equal variance t-statistic is no longer equal to the unequal variance t-statistic:

```
>>> rvs4 = stats.norm.rvs(loc=5, scale=20, size=100)
>>> stats.ttest_ind(rvs1, rvs4)
(-0.99882539442782481, 0.3182832709103896)
>>> stats.ttest_ind(rvs1, rvs4, equal_var = False)
(-0.69712570584654099, 0.48716927725402048)
```

T-test with different means, variance, and n:

```
>>> rvs5 = stats.norm.rvs(loc=8, scale=20, size=100)
>>> stats.ttest_ind(rvs1, rvs5)
(-1.4679669854490653, 0.14263895620529152)
>>> stats.ttest_ind(rvs1, rvs5, equal_var = False)
(-0.94365973617132992, 0.34744170334794122)
```

`scipy.stats.`**`ttest_ind_from_stats`**(*mean1*, *std1*, *nobs1*, *mean2*, *std2*, *nobs2*, *equal_var=True*)

T-test for means of two independent samples from descriptive statistics.

This is a two-sided test for the null hypothesis that 2 independent samples have identical average (expected) values.

Parameters

mean1 : array_like
: The mean(s) of sample 1.

std1 : array_like
: The standard deviation(s) of sample 1.

nobs1 : array_like
: The number(s) of observations of sample 1.

mean2 : array_like
: The mean(s) of sample 2

std2 : array_like
The standard deviations(s) of sample 2.
nobs2 : array_like
The number(s) of observations of sample 2.
equal_var : bool, optional
If True (default), perform a standard independent 2 sample test that assumes equal population variances *[R645]*. If False, perform Welch's t-test, which does not assume equal population variance *[R646]*.

Returns **statistic** : float or array
The calculated t-statistics
pvalue : float or array
The two-tailed p-value.

See also:

scipy.stats.ttest_ind

Notes

New in version 0.16.0.

References

[R645], *[R646]*

scipy.stats.**ttest_rel**(*a*, *b*, *axis=0*, *nan_policy='propagate'*)

Calculates the T-test on TWO RELATED samples of scores, a and b.

This is a two-sided test for the null hypothesis that 2 related or repeated samples have identical average (expected) values.

Parameters **a, b** : array_like
The arrays must have the same shape.
axis : int or None, optional
Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.
nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns **statistic** : float or array
t-statistic
pvalue : float or array
two-tailed p-value

Notes

Examples for the use are scores of the same set of student in different exams, or repeated sampling from the same units. The test measures whether the average score differs significantly across samples (e.g. exams). If we observe a large p-value, for example greater than 0.05 or 0.1 then we cannot reject the null hypothesis of identical average scores. If the p-value is smaller than the threshold, e.g. 1%, 5% or 10%, then we reject the null hypothesis of equal averages. Small p-values are associated with large t-statistics.

References

http://en.wikipedia.org/wiki/T-test#Dependent_t-test

Examples

```
>>> from scipy import stats
>>> np.random.seed(12345678) # fix random seed to get same numbers
```

```
>>> rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
>>> rvs2 = (stats.norm.rvs(loc=5,scale=10,size=500) +
...         stats.norm.rvs(scale=0.2,size=500))
>>> stats.ttest_rel(rvs1,rvs2)
(0.24101764965300962, 0.80964043445811562)
>>> rvs3 = (stats.norm.rvs(loc=8,scale=10,size=500) +
...         stats.norm.rvs(scale=0.2,size=500))
>>> stats.ttest_rel(rvs1,rvs3)
(-3.9995108708727933, 7.3082402191726459e-005)
```

`scipy.stats.`**`kstest`**(*rvs*, *cdf*, *args=()*, *N=20*, *alternative='two-sided'*, *mode='approx'*)

Perform the Kolmogorov-Smirnov test for goodness of fit.

This performs a test of the distribution G(x) of an observed random variable against a given distribution F(x). Under the null hypothesis the two distributions are identical, G(x)=F(x). The alternative hypothesis can be either 'two-sided' (default), 'less' or 'greater'. The KS test is only valid for continuous distributions.

Parameters

rvs : str, array or callable

If a string, it should be the name of a distribution in `scipy.stats`. If an array, it should be a 1-D array of observations of random variables. If a callable, it should be a function to generate random variables; it is required to have a keyword argument *size*.

cdf : str or callable

If a string, it should be the name of a distribution in `scipy.stats`. If *rvs* is a string then *cdf* can be False or the same as *rvs*. If a callable, that callable is used to calculate the cdf.

args : tuple, sequence, optional

Distribution parameters, used if *rvs* or *cdf* are strings.

N : int, optional

Sample size if *rvs* is string or callable. Default is 20.

alternative : {'two-sided', 'less','greater'}, optional

Defines the alternative hypothesis (see explanation above). Default is 'two-sided'.

mode : 'approx' (default) or 'asymp', optional

Defines the distribution used for calculating the p-value.

- 'approx' : use approximation to exact distribution of test statistic
- 'asymp' : use asymptotic distribution of test statistic

Returns

statistic : float

KS test statistic, either D, D+ or D-.

pvalue : float

One-tailed or two-tailed p-value.

Notes

In the one-sided test, the alternative is that the empirical cumulative distribution function of the random variable is "less" or "greater" than the cumulative distribution function F(x) of the hypothesis, `G(x)<=F(x)`, resp. `G(x)>=F(x)`.

Examples

```
>>> from scipy import stats
```

```
>>> x = np.linspace(-15, 15, 9)
>>> stats.kstest(x, 'norm')
(0.44435602715924361, 0.038850142705171065)
```

```
>>> np.random.seed(987654321) # set random seed to get the same result
>>> stats.kstest('norm', False, N=100)
(0.058352892479417884, 0.88531190944151261)
```

The above lines are equivalent to:

```
>>> np.random.seed(987654321)
>>> stats.kstest(stats.norm.rvs(size=100), 'norm')
(0.058352892479417884, 0.88531190944151261)
```

Test against one-sided alternative hypothesis

Shift distribution to larger values, so that `cdf_dgp(x) < norm.cdf(x)`:

```
>>> np.random.seed(987654321)
>>> x = stats.norm.rvs(loc=0.2, size=100)
>>> stats.kstest(x,'norm', alternative = 'less')
(0.12464329735846891, 0.040989164077641749)
```

Reject equal distribution against alternative hypothesis: less

```
>>> stats.kstest(x,'norm', alternative = 'greater')
(0.0072115233216311081, 0.98531158590396395)
```

Don't reject equal distribution against alternative hypothesis: greater

```
>>> stats.kstest(x,'norm', mode='asymp')
(0.12464329735846891, 0.08944488871182088)
```

Testing t distributed random variables against normal distribution

With 100 degrees of freedom the t distribution looks close to the normal distribution, and the K-S test does not reject the hypothesis that the sample came from the normal distribution:

```
>>> np.random.seed(987654321)
>>> stats.kstest(stats.t.rvs(100,size=100),'norm')
(0.072018929165471257, 0.67630062862479168)
```

With 3 degrees of freedom the t distribution looks sufficiently different from the normal distribution, that we can reject the hypothesis that the sample came from the normal distribution at the 10% level:

```
>>> np.random.seed(987654321)
>>> stats.kstest(stats.t.rvs(3,size=100),'norm')
(0.131016895759829, 0.058826222555312224)
```

`scipy.stats.`**`chisquare`**(*f_obs*, *f_exp=None*, *ddof=0*, *axis=0*)

Calculates a one-way chi square test.

The chi square test tests the null hypothesis that the categorical data has the given frequencies.

Parameters **f_obs** : array_like

Observed frequencies in each category.

f_exp : array_like, optional

Expected frequencies in each category. By default the categories are assumed to be equally likely.

ddof : int, optional

"Delta degrees of freedom": adjustment to the degrees of freedom for the p-value. The p-value is computed using a chi-squared distribution with `k - 1 - ddof` degrees of freedom, where *k* is the number of observed frequencies. The default value of *ddof* is 0.

axis : int or None, optional

The axis of the broadcast result of *f_obs* and *f_exp* along which to apply the test. If axis is None, all values in *f_obs* are treated as a single data set. Default is 0.

Returns **chisq** : float or ndarray

The chi-squared test statistic. The value is a float if *axis* is None or *f_obs* and *f_exp* are 1-D.

p : float or ndarray

The p-value of the test. The value is a float if *ddof* and the return value *chisq* are scalars.

See also:

`power_divergence`, `mstats.chisquare`

Notes

This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5.

The default degrees of freedom, k-1, are for the case when no parameters of the distribution are estimated. If p parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are k-1-p. If the parameters are estimated in a different way, then the dof can be between k-1-p and k-1. However, it is also possible that the asymptotic distribution is not a chisquare, in which case this test is not appropriate.

References

[R567], *[R568]*

Examples

When just *f_obs* is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies.

```
>>> from scipy.stats import chisquare
>>> chisquare([16, 18, 16, 14, 12, 12])
(2.0, 0.84914503608460956)
```

With *f_exp* the expected frequencies can be given.

```
>>> chisquare([16, 18, 16, 14, 12, 12], f_exp=[16, 16, 16, 16, 16, 8])
(3.5, 0.62338762774958223)
```

When *f_obs* is 2-D, by default the test is applied to each column.

```
>>> obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20, 24]]).T
>>> obs.shape
(6, 2)
>>> chisquare(obs)
(array([ 2.        ,  6.66666667]), array([ 0.84914504,  0.24663415]))
```

By setting `axis=None`, the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```
>>> chisquare(obs, axis=None)
(23.31034482758621, 0.015975692534127565)
>>> chisquare(obs.ravel())
(23.31034482758621, 0.015975692534127565)
```

ddof is the change to make to the default degrees of freedom.

```
>>> chisquare([16, 18, 16, 14, 12, 12], ddof=1)
(2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the chi-squared statistic with *ddof*.

```
>>> chisquare([16, 18, 16, 14, 12, 12], ddof=[0,1,2])
(2.0, array([ 0.84914504,  0.73575888,  0.5724067 ]))
```

f_obs and *f_exp* are also broadcast. In the following, *f_obs* has shape (6,) and *f_exp* has shape (2, 6), so the result of broadcasting *f_obs* and *f_exp* has shape (2, 6). To compute the desired chi-squared statistics, we use `axis=1`:

```
>>> chisquare([16, 18, 16, 14, 12, 12],
...           f_exp=[[16, 16, 16, 16, 16, 8], [8, 20, 20, 16, 12, 12]],
...           axis=1)
(array([ 3.5 ,  9.25]), array([ 0.62338763,  0.09949846]))
```

`scipy.stats.`**`power_divergence`**(*f_obs*, *f_exp=None*, *ddof=0*, *axis=0*, *lambda_=None*)

Cressie-Read power divergence statistic and goodness of fit test.

This function tests the null hypothesis that the categorical data has the given frequencies, using the Cressie-Read power divergence statistic.

Parameters

f_obs : array_like
: Observed frequencies in each category.

f_exp : array_like, optional
: Expected frequencies in each category. By default the categories are assumed to be equally likely.

ddof : int, optional
: "Delta degrees of freedom": adjustment to the degrees of freedom for the p-value. The p-value is computed using a chi-squared distribution with `k - 1 - ddof` degrees of freedom, where *k* is the number of observed frequencies. The default value of *ddof* is 0.

axis : int or None, optional
: The axis of the broadcast result of *f_obs* and *f_exp* along which to apply the test. If axis is None, all values in *f_obs* are treated as a single data set. Default is 0.

lambda_ : float or str, optional
: *lambda_* gives the power in the Cressie-Read power divergence statistic. The default is 1. For convenience, *lambda_* may be assigned one of the following strings, in which case the corresponding numerical value is used:

```
String              Value   Description
"pearson"             1     Pearson's chi-squared␣
↪statistic.
                            In this case, the function␣
↪is
```

```
                                equivalent to `stats.
↪chisquare`.
"log-likelihood"      0     Log-likelihood ratio. Also␣
↪known as
                                the G-test [R623]_.
"freeman-tukey"      -1/2   Freeman-Tukey statistic.
"mod-log-likelihood" -1     Modified log-likelihood␣
↪ratio.
"neyman"             -2     Neyman's statistic.
"cressie-read"        2/3   The power recommended in␣
↪[R625]_.
```

Returns **statistic** : float or ndarray

The Cressie-Read power divergence test statistic. The value is a float if *axis* is None or if‘ *f_obs* and *f_exp* are 1-D.

pvalue : float or ndarray

The p-value of the test. The value is a float if *ddof* and the return value `stat` are scalars.

See also:

chisquare

Notes

This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5.

When *lambda_* is less than zero, the formula for the statistic involves dividing by *f_obs*, so a warning or error may be generated if any value in *f_obs* is 0.

Similarly, a warning or error may be generated if any value in *f_exp* is zero when *lambda_* >= 0.

The default degrees of freedom, k-1, are for the case when no parameters of the distribution are estimated. If p parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are k-1-p. If the parameters are estimated in a different way, then the dof can be between k-1-p and k-1. However, it is also possible that the asymptotic distribution is not a chisquare, in which case this test is not appropriate.

This function handles masked arrays. If an element of *f_obs* or *f_exp* is masked, then data at that position is ignored, and does not count towards the size of the data set.

New in version 0.13.0.

References

[R621], *[R622]*, *[R623]*, *[R624]*, *[R625]*

Examples

(See *chisquare* for more examples.)

When just *f_obs* is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies. Here we perform a G-test (i.e. use the log-likelihood ratio statistic):

```
>>> from scipy.stats import power_divergence
>>> power_divergence([16, 18, 16, 14, 12, 12], lambda_='log-likelihood')
(2.006573162632538, 0.84823476779463769)
```

The expected frequencies can be given with the *f_exp* argument:

```
>>> power_divergence([16, 18, 16, 14, 12, 12],
...                  f_exp=[16, 16, 16, 16, 16, 8],
...                  lambda_='log-likelihood')
(3.3281031458963746, 0.6495419288047497)
```

When *f_obs* is 2-D, by default the test is applied to each column.

```
>>> obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20, 24]]).T
>>> obs.shape
(6, 2)
>>> power_divergence(obs, lambda_="log-likelihood")
(array([ 2.00657316,  6.77634498]), array([ 0.84823477,  0.23781225]))
```

By setting `axis=None`, the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```
>>> power_divergence(obs, axis=None)
(23.31034482758621, 0.015975692534127565)
>>> power_divergence(obs.ravel())
(23.31034482758621, 0.015975692534127565)
```

ddof is the change to make to the default degrees of freedom.

```
>>> power_divergence([16, 18, 16, 14, 12, 12], ddof=1)
(2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the test statistic with *ddof*.

```
>>> power_divergence([16, 18, 16, 14, 12, 12], ddof=[0,1,2])
(2.0, array([ 0.84914504,  0.73575888,  0.5724067 ]))
```

f_obs and *f_exp* are also broadcast. In the following, *f_obs* has shape (6,) and *f_exp* has shape (2, 6), so the result of broadcasting *f_obs* and *f_exp* has shape (2, 6). To compute the desired chi-squared statistics, we must use `axis=1`:

```
>>> power_divergence([16, 18, 16, 14, 12, 12],
...                  f_exp=[[16, 16, 16, 16, 16, 8],
...                         [8, 20, 20, 16, 12, 12]],
...                  axis=1)
(array([ 3.5 ,  9.25]), array([ 0.62338763,  0.09949846]))
```

`scipy.stats.`**`ks_2samp`**(*data1*, *data2*)

Computes the Kolmogorov-Smirnov statistic on 2 samples.

This is a two-sided test for the null hypothesis that 2 independent samples are drawn from the same continuous distribution.

Parameters

data1, data2 : sequence of 1-D ndarrays
two arrays of sample observations assumed to be drawn from a continuous distribution, sample sizes can be different

Returns

statistic : float
KS statistic

pvalue : float
two-tailed p-value

Notes

This tests whether 2 samples are drawn from the same distribution. Note that, like in the case of the one-sample K-S test, the distribution is assumed to be continuous.

This is the two-sided test, one-sided tests are not implemented. The test uses the two-sided asymptotic Kolmogorov-Smirnov distribution.

If the K-S statistic is small or the p-value is high, then we cannot reject the hypothesis that the distributions of the two samples are the same.

Examples

```
>>> from scipy import stats
>>> np.random.seed(12345678)  #fix random seed to get the same result
>>> n1 = 200  # size of first sample
>>> n2 = 300  # size of second sample
```

For a different distribution, we can reject the null hypothesis since the pvalue is below 1%:

```
>>> rvs1 = stats.norm.rvs(size=n1, loc=0., scale=1)
>>> rvs2 = stats.norm.rvs(size=n2, loc=0.5, scale=1.5)
>>> stats.ks_2samp(rvs1, rvs2)
(0.20833333333333337, 4.6674975515806989e-005)
```

For a slightly different distribution, we cannot reject the null hypothesis at a 10% or lower alpha since the p-value at 0.144 is higher than 10%

```
>>> rvs3 = stats.norm.rvs(size=n2, loc=0.01, scale=1.0)
>>> stats.ks_2samp(rvs1, rvs3)
(0.10333333333333333, 0.14498781825751686)
```

For an identical distribution, we cannot reject the null hypothesis since the p-value is high, 41%:

```
>>> rvs4 = stats.norm.rvs(size=n2, loc=0.0, scale=1.0)
>>> stats.ks_2samp(rvs1, rvs4)
(0.07999999999999996, 0.41126949729859719)
```

`scipy.stats.`**`mannwhitneyu`**(*x*, *y*, *use_continuity=True*, *alternative=None*)

Computes the Mann-Whitney rank test on samples x and y.

Parameters

x, y : array_like
: Array of samples, should be one-dimensional.

use_continuity : bool, optional
: Whether a continuity correction (1/2.) should be taken into account. Default is True.

alternative : None (deprecated), 'less', 'two-sided', or 'greater'
: Whether to get the p-value for the one-sided hypothesis ('less' or 'greater') or for the two-sided hypothesis ('two-sided'). Defaults to None, which results in a p-value half the size of the 'two-sided' p-value and a different U statistic. The default behavior is not the same as using 'less' or 'greater': it only exists for backward compatibility and is deprecated.

Returns

statistic : float
: The Mann-Whitney U statistic, equal to min(U for x, U for y) if *alternative* is equal to None (deprecated; exists for backward compatibility), and U for y otherwise.

pvalue : float
: p-value assuming an asymptotic normal distribution. One-sided or two-sided, depending on the choice of *alternative*.

Notes

Use only when the number of observation in each sample is > 20 and you have 2 independent samples of ranks. Mann-Whitney U is significant if the u-obtained is LESS THAN or equal to the critical value of U.

This test corrects for ties and by default uses a continuity correction.

`scipy.stats.tiecorrect(rankvals)`

Tie correction factor for ties in the Mann-Whitney U and Kruskal-Wallis H tests.

Parameters **rankvals** : array_like

A 1-D sequence of ranks. Typically this will be the array returned by `stats.rankdata`.

Returns **factor** : float

Correction factor for U or H.

See also:

rankdata Assign ranks to the data

mannwhitneyu Mann-Whitney rank test

kruskal Kruskal-Wallis H test

References

[R642]

Examples

```
>>> from scipy.stats import tiecorrect, rankdata
>>> tiecorrect([1, 2.5, 2.5, 4])
0.9
>>> ranks = rankdata([1, 3, 2, 4, 5, 7, 2, 8, 4])
>>> ranks
array([ 1. ,  4. ,  2.5,  5.5,  7. ,  8. ,  2.5,  9. ,  5.5])
>>> tiecorrect(ranks)
0.9833333333333333
```

`scipy.stats.rankdata(a, method='average')`

Assign ranks to data, dealing with ties appropriately.

Ranks begin at 1. The *method* argument controls how ranks are assigned to equal values. See *[R629]* for further discussion of ranking methods.

Parameters **a** : array_like

The array of values to be ranked. The array is first flattened.

method : str, optional

The method used to assign ranks to tied elements. The options are 'average', 'min', 'max', 'dense' and 'ordinal'.

'average': The average of the ranks that would have been assigned to all the tied values is assigned to each value.

'min': The minimum of the ranks that would have been assigned to all the tied values is assigned to each value. (This is also referred to as "competition" ranking.)

'max': The maximum of the ranks that would have been assigned to all the tied values is assigned to each value.

'dense': Like 'min', but the rank of the next highest element is assigned the rank immediately after those assigned to the tied elements.

'ordinal': All values are given a distinct rank, corresponding to the order that the values occur in *a*.

The default is 'average'.

Returns **ranks** : ndarray
An array of length equal to the size of *a*, containing rank scores.

References

[R629]

Examples

```
>>> from scipy.stats import rankdata
>>> rankdata([0, 2, 3, 2])
array([ 1. ,  2.5,  4. ,  2.5])
>>> rankdata([0, 2, 3, 2], method='min')
array([ 1,  2,  4,  2])
>>> rankdata([0, 2, 3, 2], method='max')
array([ 1,  3,  4,  3])
>>> rankdata([0, 2, 3, 2], method='dense')
array([ 1,  2,  3,  2])
>>> rankdata([0, 2, 3, 2], method='ordinal')
array([ 1,  2,  4,  3])
```

`scipy.stats.`**`ranksums`**(*x*, *y*)

Compute the Wilcoxon rank-sum statistic for two samples.

The Wilcoxon rank-sum test tests the null hypothesis that two sets of measurements are drawn from the same distribution. The alternative hypothesis is that values in one sample are more likely to be larger than the values in the other sample.

This test should be used to compare two samples from continuous distributions. It does not handle ties between measurements in x and y. For tie-handling and an optional continuity correction see `scipy.stats.mannwhitneyu`.

Parameters **x,y** : array_like
The data from the two samples

Returns **statistic** : float
The test statistic under the large-sample approximation that the rank sum statistic is normally distributed

pvalue : float
The two-sided p-value of the test

References

[R630]

`scipy.stats.`**`wilcoxon`**(*x*, *y=None*, *zero_method='wilcox'*, *correction=False*)

Calculate the Wilcoxon signed-rank test.

The Wilcoxon signed-rank test tests the null hypothesis that two related paired samples come from the same distribution. In particular, it tests whether the distribution of the differences x - y is symmetric about zero. It is a non-parametric version of the paired T-test.

Parameters **x** : array_like
The first set of measurements.

y : array_like, optional
The second set of measurements. If *y* is not given, then the *x* array is considered to be the differences between the two sets of measurements.

zero_method : string, {"pratt", "wilcox", "zsplit"}, optional
"pratt": Pratt treatment: includes zero-differences in the ranking process (more conservative)
"wilcox": Wilcox treatment: discards all zero-differences

"zsplit": Zero rank split: just like Pratt, but spliting the zero rank between positive and negative ones

correction : bool, optional
If True, apply continuity correction by adjusting the Wilcoxon rank statistic by 0.5 towards the mean value when computing the z-statistic. Default is False.

Returns

statistic : float
The sum of the ranks of the differences above or below zero, whichever is smaller.

pvalue : float
The two-sided p-value for the test.

Notes

Because the normal approximation is used for the calculations, the samples used should be large. A typical rule is to require that n > 20.

References

[R651]

`scipy.stats.kruskal(*args, **kwargs)`

Compute the Kruskal-Wallis H-test for independent samples

The Kruskal-Wallis H-test tests the null hypothesis that the population median of all of the groups are equal. It is a non-parametric version of ANOVA. The test works on 2 or more independent samples, which may have different sizes. Note that rejecting the null hypothesis does not indicate which of the groups differs. Post-hoc comparisons between groups are required to determine which groups are different.

Parameters

sample1, sample2, ... : array_like
Two or more arrays with the sample measurements can be given as arguments.

nan_policy : {'propagate', 'raise', 'omit'}, optional
Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns

statistic : float
The Kruskal-Wallis H statistic, corrected for ties

pvalue : float
The p-value for the test using the assumption that H has a chi square distribution

See also:

`f_oneway` 1-way ANOVA

`mannwhitneyu`
Mann-Whitney rank test on two samples.

`friedmanchisquare`
Friedman test for repeated measurements

Notes

Due to the assumption that H has a chi square distribution, the number of samples in each group must not be too small. A typical rule is that each sample must have at least 5 measurements.

References

[R599], *[R600]*

Examples

```
>>> from scipy import stats
>>> x = [1, 3, 5, 7, 9]
>>> y = [2, 4, 6, 8, 10]
>>> stats.kruskal(x, y)
KruskalResult(statistic=0.27272727272727337, pvalue=0.60150813444058948)
```

```
>>> x = [1, 1, 1]
>>> y = [2, 2, 2]
>>> z = [2, 2]
>>> stats.kruskal(x, y, z)
KruskalResult(statistic=7.0, pvalue=0.030197383422318501)
```

`scipy.stats.`**`friedmanchisquare`**`(*args)`

Computes the Friedman test for repeated measurements

The Friedman test tests the null hypothesis that repeated measurements of the same individuals have the same distribution. It is often used to test for consistency among measurements obtained in different ways. For example, if two measurement techniques are used on the same set of individuals, the Friedman test can be used to determine if the two measurement techniques are consistent.

Parameters **measurements1, measurements2, measurements3...** : array_like

Arrays of measurements. All of the arrays must have the same number of elements. At least 3 sets of measurements must be given.

Returns **statistic** : float

the test statistic, correcting for ties

pvalue : float

the associated p-value assuming that the test statistic has a chi squared distribution

Notes

Due to the assumption that the test statistic has a chi squared distribution, the p-value is only reliable for n > 10 and more than 6 repeated measurements.

References

[R582]

`scipy.stats.`**`combine_pvalues`**`(pvalues, method='fisher', weights=None)`

Methods for combining the p-values of independent tests bearing upon the same hypothesis.

Parameters **pvalues** : array_like, 1-D

Array of p-values assumed to come from independent tests.

method : {'fisher', 'stouffer'}, optional

Name of method to use to combine p-values. The following methods are available:

- "fisher": Fisher's method (Fisher's combined probability test), the default
- "stouffer": Stouffer's Z-score method.

weights : array_like, 1-D, optional

Optional array of weights used only for Stouffer's Z-score method.

Returns statistic: float

The statistic calculated by the specified method: - "fisher": The chi-squared statistic - "stouffer": The Z-score

pval: float

The combined p-value.

Notes

Fisher's method (also known as Fisher's combined probability test) *[R569]* uses a chi-squared statistic to compute a combined p-value. The closely related Stouffer's Z-score method *[R570]* uses Z-scores rather than p-values. The advantage of Stouffer's method is that it is straightforward to introduce weights, which can make Stouffer's method more powerful than Fisher's method when the p-values are from studies of different size *[R571]* *[R572]*.

Fisher's method may be extended to combine p-values from dependent tests *[R573]*. Extensions such as Brown's method and Kost's method are not currently implemented.

New in version 0.15.0.

References

[R569], *[R570]*, *[R571]*, *[R572]*, *[R573]*

`scipy.stats.ss(*args, **kwds)`

`ss` is deprecated! scipy.stats.ss is deprecated in scipy 0.17.0

`scipy.stats.square_of_sums(*args, **kwds)`

`square_of_sums` is deprecated! scipy.stats.square_of_sums is deprecated in scipy 0.17.0

`scipy.stats.jarque_bera(x)`

Perform the Jarque-Bera goodness of fit test on sample data.

The Jarque-Bera test tests whether the sample data has the skewness and kurtosis matching a normal distribution.

Note that this test only works for a large enough number of data samples (>2000) as the test statistic asymptotically has a Chi-squared distribution with 2 degrees of freedom.

Parameters

x : array_like

Observations of a random variable.

Returns

jb_value : float

The test statistic.

p : float

The p-value for the hypothesis test.

References

[R594]

Examples

```
>>> from scipy import stats
>>> np.random.seed(987654321)
>>> x = np.random.normal(0, 1, 100000)
>>> y = np.random.rayleigh(1, 100000)
>>> stats.jarque_bera(x)
(4.7165707989581342, 0.09458225503041906)
>>> stats.jarque_bera(y)
(6713.7098548143422, 0.0)
```

`ansari(x, y)`	Perform the Ansari-Bradley test for equal scale parameters
`bartlett(*args)`	Perform Bartlett's test for equal variances
`levene(*args, **kwds)`	Perform Levene test for equal variances.
`shapiro(x[, a, reta])`	Perform the Shapiro-Wilk test for normality.
`anderson(x[, dist])`	Anderson-Darling test for data coming from a particular distribution

Continued on next page

Table 5.284 – continued from previous page

anderson_ksamp(samples[, midrank])	The Anderson-Darling test for k-samples.
binom_test(x[, n, p, alternative])	Perform a test that the probability of success is p.
fligner(*args, **kwds)	Perform Fligner-Killeen test for equality of variance.
median_test(*args, **kwds)	Mood's median test.
mood(x, y[, axis])	Perform Mood's test for equal scale parameters.

scipy.stats.**ansari**(*x*, *y*)

Perform the Ansari-Bradley test for equal scale parameters

The Ansari-Bradley test is a non-parametric test for the equality of the scale parameter of the distributions from which two samples were drawn.

Parameters **x, y** : array_like

arrays of sample data

Returns **statistic** : float

The Ansari-Bradley test statistic

pvalue : float

The p-value of the hypothesis test

See also:

fligner A non-parametric test for the equality of k variances

mood A non-parametric test for the equality of two scale parameters

Notes

The p-value given is exact when the sample sizes are both less than 55 and there are no ties, otherwise a normal approximation for the p-value is used.

References

[R554]

scipy.stats.**bartlett**(**args*)

Perform Bartlett's test for equal variances

Bartlett's test tests the null hypothesis that all input samples are from populations with equal variances. For samples from significantly non-normal populations, Levene's test *levene* is more robust.

Parameters **sample1, sample2,...** : array_like

arrays of sample data. May be different lengths.

Returns **statistic** : float

The test statistic.

pvalue : float

The p-value of the test.

See also:

fligner A non-parametric test for the equality of k variances

levene A robust parametric test for equality of k variances

Notes

Conover et al. (1981) examine many of the existing parametric and nonparametric tests by extensive simulations and they conclude that the tests proposed by Fligner and Killeen (1976) and Levene (1960) appear to be superior in terms of robustness of departures from normality and power *[R558]*.

References

[R556], [R557], [R558], [R559]

`scipy.stats.levene(*args, **kwds)`

Perform Levene test for equal variances.

The Levene test tests the null hypothesis that all input samples are from populations with equal variances. Levene's test is an alternative to Bartlett's test `bartlett` in the case where there are significant deviations from normality.

Parameters

sample1, sample2, ... : array_like
: The sample data, possibly with different lengths

center : {'mean', 'median', 'trimmed'}, optional
: Which function of the data to use in the test. The default is 'median'.

proportiontocut : float, optional
: When *center* is 'trimmed', this gives the proportion of data points to cut from each end. (See `scipy.stats.trim_mean`.) Default is 0.05.

Returns

statistic : float
: The test statistic.

pvalue : float
: The p-value for the test.

Notes

Three variations of Levene's test are possible. The possibilities and their recommended usages are:

- 'median' : Recommended for skewed (non-normal) distributions>
- 'mean' : Recommended for symmetric, moderate-tailed distributions.
- 'trimmed' : Recommended for heavy-tailed distributions.

References

[R603], [R604], [R605]

`scipy.stats.shapiro(x, a=None, reta=False)`

Perform the Shapiro-Wilk test for normality.

The Shapiro-Wilk test tests the null hypothesis that the data was drawn from a normal distribution.

Parameters

x : array_like
: Array of sample data.

a : array_like, optional
: Array of internal parameters used in the calculation. If these are not given, they will be computed internally. If x has length n, then a must have length n/2.

reta : bool, optional
: Whether or not to return the internally computed a values. The default is False.

Returns

W : float
: The test statistic.

p-value : float
: The p-value for the hypothesis test.

a : array_like, optional
: If *reta* is True, then these are the internally computed "a" values that may be passed into this function on future calls.

See also:

`anderson` The Anderson-Darling test for normality

`kstest` The Kolmogorov-Smirnov test for goodness of fit.

Notes

The algorithm used is described in *[R634]* but censoring parameters as described are not implemented. For N > 5000 the W test statistic is accurate but the p-value may not be.

The chance of rejecting the null hypothesis when it is true is close to 5% regardless of sample size.

References

[R631], *[R632]*, *[R633]*, *[R634]*

Examples

```
>>> from scipy import stats
>>> np.random.seed(12345678)
>>> x = stats.norm.rvs(loc=5, scale=3, size=100)
>>> stats.shapiro(x)
(0.9772805571556091, 0.08144091814756393)
```

`scipy.stats.`**`anderson`**(*x*, *dist='norm'*)

Anderson-Darling test for data coming from a particular distribution

The Anderson-Darling test is a modification of the Kolmogorov- Smirnov test `kstest` for the null hypothesis that a sample is drawn from a population that follows a particular distribution. For the Anderson-Darling test, the critical values depend on which distribution is being tested against. This function works for normal, exponential, logistic, or Gumbel (Extreme Value Type I) distributions.

Parameters

x : array_like
  array of sample data

dist : {'norm','expon','logistic','gumbel','gumbel_l', gumbel_r', 'extreme1'}, optional the type of distribution to test against. The default is 'norm' and 'extreme1', 'gumbel_l' and 'gumbel' are synonyms.

Returns

statistic : float
  The Anderson-Darling test statistic

critical_values : list
  The critical values for this distribution

significance_level : list
  The significance levels for the corresponding critical values in percents. The function returns critical values for a differing set of significance levels depending on the distribution that is being tested against.

Notes

Critical values provided are for the following significance levels:

normal/exponenential
  15%, 10%, 5%, 2.5%, 1%

logistic 25%, 10%, 5%, 2.5%, 1%, 0.5%

Gumbel 25%, 10%, 5%, 2.5%, 1%

If A2 is larger than these critical values then for the corresponding significance level, the null hypothesis that the data come from the chosen distribution can be rejected.

References

[R547], *[R548]*, *[R549]*, *[R550]*, *[R551]*, *[R552]*

`scipy.stats.`**`anderson_ksamp`**(*samples*, *midrank=True*)

The Anderson-Darling test for k-samples.

The k-sample Anderson-Darling test is a modification of the one-sample Anderson-Darling test. It tests the null hypothesis that k-samples are drawn from the same population without having to specify the distribution function of that population. The critical values depend on the number of samples.

Parameters

samples : sequence of 1-D array_like
Array of sample data in arrays.

midrank : bool, optional
Type of Anderson-Darling test which is computed. Default (True) is the midrank test applicable to continuous and discrete populations. If False, the right side empirical distribution is used.

Returns

statistic : float
Normalized k-sample Anderson-Darling test statistic.

critical_values : array
The critical values for significance levels 25%, 10%, 5%, 2.5%, 1%.

significance_level : float
An approximate significance level at which the null hypothesis for the provided samples can be rejected.

Raises

ValueError
If less than 2 samples are provided, a sample is empty, or no distinct observations are in the samples.

See also:

`ks_2samp` 2 sample Kolmogorov-Smirnov test
`anderson` 1 sample Anderson-Darling test

Notes

[R553] Defines three versions of the k-sample Anderson-Darling test: one for continuous distributions and two for discrete distributions, in which ties between samples may occur. The default of this routine is to compute the version based on the midrank empirical distribution function. This test is applicable to continuous and discrete data. If midrank is set to False, the right side empirical distribution is used for a test for discrete data. According to *[R553]*, the two discrete test statistics differ only slightly if a few collisions due to round-off errors occur in the test not adjusted for ties between samples.

New in version 0.14.0.

References

[R553]

Examples

```
>>> from scipy import stats
>>> np.random.seed(314159)
```

The null hypothesis that the two random samples come from the same distribution can be rejected at the 5% level because the returned test value is greater than the critical value for 5% (1.961) but not at the 2.5% level. The interpolation gives an approximate significance level of 3.1%:

```
>>> stats.anderson_ksamp([np.random.normal(size=50),
... np.random.normal(loc=0.5, size=30)])
(2.4615796189876105,
  array([ 0.325,  1.226,  1.961,  2.718,  3.752]),
  0.03134990135800783)
```

The null hypothesis cannot be rejected for three samples from an identical distribution. The approximate p-value (87%) has to be computed by extrapolation and may not be very accurate:

```
>>> stats.anderson_ksamp([np.random.normal(size=50),
... np.random.normal(size=30), np.random.normal(size=20)])
(-0.73091722665244196,
  array([ 0.44925884,  1.3052767 ,  1.9434184 ,  2.57696569,  3.41634856]),
  0.8789283903979661)
```

`scipy.stats.`**`binom_test`**(*x*, *n=None*, *p=0.5*, *alternative='two-sided'*)

Perform a test that the probability of success is p.

This is an exact, two-sided test of the null hypothesis that the probability of success in a Bernoulli experiment is *p*.

Parameters **x** : integer or array_like

the number of successes, or if x has length 2, it is the number of successes and the number of failures.

n : integer

the number of trials. This is ignored if x gives both the number of successes and failures

p : float, optional

The hypothesized probability of success. 0 <= p <= 1. The default value is p = 0.5

alternative : {'two-sided', 'greater', 'less'}, optional

Indicates the alternative hypothesis. The default value is 'two-sided'.

Returns **p-value** : float

The p-value of the hypothesis test

References

[R560]

`scipy.stats.`**`fligner`**(**args*, ***kwds*)

Perform Fligner-Killeen test for equality of variance.

Fligner's test tests the null hypothesis that all input samples are from populations with equal variances. Fligner-Killeen's test is distribution free when populations are identical *[R579]*.

Parameters **sample1, sample2, ...** : array_like

Arrays of sample data. Need not be the same length.

center : {'mean', 'median', 'trimmed'}, optional

Keyword argument controlling which function of the data is used in computing the test statistic. The default is 'median'.

proportiontocut : float, optional

When *center* is 'trimmed', this gives the proportion of data points to cut from each end. (See `scipy.stats.trim_mean`.) Default is 0.05.

Returns **statistic** : float

The test statistic.

pvalue : float

The p-value for the hypothesis test.

See also:

`bartlett` A parametric test for equality of k variances in normal samples

`levene` A robust parametric test for equality of k variances

Notes

As with Levene's test there are three variants of Fligner's test that differ by the measure of central tendency used in the test. See `levene` for more information.

Conover et al. (1981) examine many of the existing parametric and nonparametric tests by extensive simulations and they conclude that the tests proposed by Fligner and Killeen (1976) and Levene (1960) appear to be superior in terms of robustness of departures from normality and power *[R580]*.

References

[R578], *[R579]*, *[R580]*, *[R581]*

`scipy.stats.median_test(*args, **kwds)`

Mood's median test.

Test that two or more samples come from populations with the same median.

Let `n = len(args)` be the number of samples. The "grand median" of all the data is computed, and a contingency table is formed by classifying the values in each sample as being above or below the grand median. The contingency table, along with *correction* and *lambda_*, are passed to `scipy.stats.chi2_contingency` to compute the test statistic and p-value.

Parameters **sample1, sample2, ...** : array_like

The set of samples. There must be at least two samples. Each sample must be a one-dimensional sequence containing at least one value. The samples are not required to have the same length.

ties : str, optional

Determines how values equal to the grand median are classified in the contingency table. The string must be one of:

```
"below":
    Values equal to the grand median are counted as
↪"below".
"above":
    Values equal to the grand median are counted as
↪"above".
"ignore":
    Values equal to the grand median are not counted.
```

The default is "below".

correction : bool, optional

If True, *and* there are just two samples, apply Yates' correction for continuity when computing the test statistic associated with the contingency table. Default is True.

lambda_ : float or str, optional.

By default, the statistic computed in this test is Pearson's chi-squared statistic. *lambda_* allows a statistic from the Cressie-Read power divergence family to be used instead. See `power_divergence` for details. Default is 1 (Pearson's chi-squared statistic).

nan_policy : {'propagate', 'raise', 'omit'}, optional

Defines how to handle when input contains nan. 'propagate' returns nan, 'raise' throws an error, 'omit' performs the calculations ignoring nan values. Default is 'propagate'.

Returns **stat** : float

The test statistic. The statistic that is returned is determined by *lambda_*. The default is Pearson's chi-squared statistic.

p : float

The p-value of the test.

m : float

The grand median.

table : ndarray

The contingency table. The shape of the table is (2, n), where n is the number of samples. The first row holds the counts of the values above the grand median, and the second row holds the counts of the values below the grand median. The table allows further analysis with, for example, *scipy.stats.chi2_contingency*, or with *scipy.stats.fisher_exact* if there are two samples, without having to recompute the table. If `nan_policy` is "propagate" and there are nans in the input, the return value for `table` is `None`.

See also:

kruskal Compute the Kruskal-Wallis H-test for independent samples.

mannwhitneyu Computes the Mann-Whitney rank test on samples x and y.

Notes

New in version 0.15.0.

References

[R607], *[R608]*

Examples

A biologist runs an experiment in which there are three groups of plants. Group 1 has 16 plants, group 2 has 15 plants, and group 3 has 17 plants. Each plant produces a number of seeds. The seed counts for each group are:

```
Group 1: 10 14 14 18 20 22 24 25 31 31 32 39 43 43 48 49
Group 2: 28 30 31 33 34 35 36 40 44 55 57 61 91 92 99
Group 3:  0  3  9 22 23 25 25 33 34 34 40 45 46 48 62 67 84
```

The following code applies Mood's median test to these samples.

```
>>> g1 = [10, 14, 14, 18, 20, 22, 24, 25, 31, 31, 32, 39, 43, 43, 48, 49]
>>> g2 = [28, 30, 31, 33, 34, 35, 36, 40, 44, 55, 57, 61, 91, 92, 99]
>>> g3 = [0, 3, 9, 22, 23, 25, 25, 33, 34, 34, 40, 45, 46, 48, 62, 67, 84]
>>> from scipy.stats import median_test
>>> stat, p, med, tbl = median_test(g1, g2, g3)
```

The median is

```
>>> med
34.0
```

and the contingency table is

```
>>> tbl
array([[ 5, 10,  7],
       [11,  5, 10]])
```

p is too large to conclude that the medians are not the same:

```
>>> p
0.12609082774093244
```

The "G-test" can be performed by passing `lambda_="log-likelihood"` to *median_test*.

```
>>> g, p, med, tbl = median_test(g1, g2, g3, lambda_="log-likelihood")
>>> p
0.12224779737117837
```

The median occurs several times in the data, so we'll get a different result if, for example, `ties="above"` is used:

```
>>> stat, p, med, tbl = median_test(g1, g2, g3, ties="above")
>>> p
0.063873276069553273
```

```
>>> tbl
array([[ 5, 11,  9],
       [11,  4,  8]])
```

This example demonstrates that if the data set is not large and there are values equal to the median, the p-value can be sensitive to the choice of *ties*.

`scipy.stats.mood(x, y, axis=0)`

Perform Mood's test for equal scale parameters.

Mood's two-sample test for scale parameters is a non-parametric test for the null hypothesis that two samples are drawn from the same distribution with the same scale parameter.

Parameters

x, y : array_like
: Arrays of sample data.

axis : int, optional
: The axis along which the samples are tested. *x* and *y* can be of different length along *axis*. If *axis* is None, *x* and *y* are flattened and the test is done on all values in the flattened arrays.

Returns

z : scalar or ndarray
: The z-score for the hypothesis test. For 1-D inputs a scalar is returned.

p-value : scalar ndarray
: The p-value for the hypothesis test.

See also:

fligner	A non-parametric test for the equality of k variances
ansari	A non-parametric test for the equality of 2 variances
bartlett	A parametric test for equality of k variances in normal samples
levene	A parametric test for equality of k variances

Notes

The data are assumed to be drawn from probability distributions `f(x)` and `f(x/s) / s` respectively, for some probability density function f. The null hypothesis is that `s == 1`.

For multi-dimensional arrays, if the inputs are of shapes `(n0, n1, n2, n3)` and `(n0, m1, n2, n3)`, then if `axis=1`, the resulting z and p values will have shape `(n0, n2, n3)`. Note that `n1` and `m1` don't have to be equal, but the other dimensions do.

Examples

```
>>> from scipy import stats
>>> np.random.seed(1234)
>>> x2 = np.random.randn(2, 45, 6, 7)
>>> x1 = np.random.randn(2, 30, 6, 7)
>>> z, p = stats.mood(x1, x2, axis=1)
```

```
>>> p.shape
(2, 6, 7)
```

Find the number of points where the difference in scale is not significant:

```
>>> (p > 0.1).sum()
74
```

Perform the test with different scales:

```
>>> x1 = np.random.randn(2, 30)
>>> x2 = np.random.randn(2, 35) * 10.0
>>> stats.mood(x1, x2, axis=1)
(array([-5.7178125 , -5.25342163]), array([  1.07904114e-08,   1.49299218e-07]))
```

boxcox(x[, lmbda, alpha])	Return a positive dataset transformed by a Box-Cox power transformation.
boxcox_normmax(x[, brack, method])	Compute optimal Box-Cox transform parameter for input data.
boxcox_llf(lmb, data)	The boxcox log-likelihood function.
entropy(pk[, qk, base])	Calculate the entropy of a distribution for given probability values.

scipy.stats.**boxcox**(*x*, *lmbda=None*, *alpha=None*)

Return a positive dataset transformed by a Box-Cox power transformation.

Parameters **x** : ndarray

Input array. Should be 1-dimensional.

lmbda : {None, scalar}, optional

If *lmbda* is not None, do the transformation for that value.

If *lmbda* is None, find the lambda that maximizes the log-likelihood function and return it as the second output argument.

alpha : {None, float}, optional

If `alpha` is not None, return the `100 * (1-alpha)%` confidence interval for *lmbda* as the third output argument. Must be between 0.0 and 1.0.

Returns **boxcox** : ndarray

Box-Cox power transformed array.

maxlog : float, optional

If the *lmbda* parameter is None, the second returned argument is the lambda that maximizes the log-likelihood function.

(min_ci, max_ci) : tuple of float, optional

If *lmbda* parameter is None and `alpha` is not None, this returned tuple of floats represents the minimum and maximum confidence limits given `alpha`.

See also:

probplot, *boxcox_normplot*, *boxcox_normmax*, *boxcox_llf*

Notes

The Box-Cox transform is given by:

```
y = (x**lmbda - 1) / lmbda,   for lmbda > 0
    log(x),                   for lmbda = 0
```

boxcox requires the input data to be positive. Sometimes a Box-Cox transformation provides a shift parameter to achieve this; *boxcox* does not. Such a shift parameter is equivalent to adding a positive constant to *x* before calling *boxcox*.

The confidence limits returned when `alpha` is provided give the interval where:

$$llf(\hat{\lambda}) - llf(\lambda) < \frac{1}{2}\chi^2(1 - \alpha, 1),$$

with `llf` the log-likelihood function and χ^2 the chi-squared function.

References

G.E.P. Box and D.R. Cox, "An Analysis of Transformations", Journal of the Royal Statistical Society B, 26, 211-252 (1964).

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

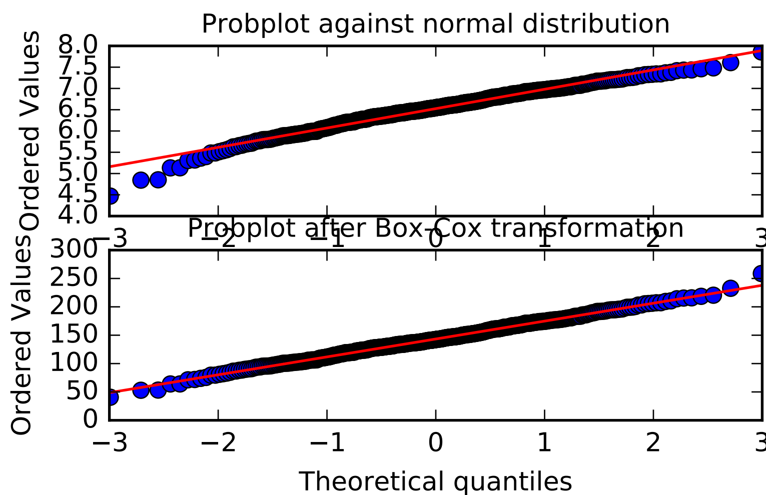
We generate some random variates from a non-normal distribution and make a probability plot for it, to show it is non-normal in the tails:

```
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(211)
>>> x = stats.loggamma.rvs(5, size=500) + 5
>>> prob = stats.probplot(x, dist=stats.norm, plot=ax1)
>>> ax1.set_xlabel('')
>>> ax1.set_title('Probplot against normal distribution')
```

We now use *boxcox* to transform the data so it's closest to normal:

```
>>> ax2 = fig.add_subplot(212)
>>> xt, _ = stats.boxcox(x)
>>> prob = stats.probplot(xt, dist=stats.norm, plot=ax2)
>>> ax2.set_title('Probplot after Box-Cox transformation')
```

```
>>> plt.show()
```



scipy.stats.**boxcox_normmax**(*x*, *brack=(-2.0, 2.0)*, *method='pearsonr'*)

Compute optimal Box-Cox transform parameter for input data.

Parameters

x : array_like

Input array.

brack : 2-tuple, optional

The starting interval for a downhill bracket search with *optimize.brent*. Note that this is in most cases not critical; the final result is allowed to be outside this bracket.

method : str, optional

The method to determine the optimal transform parameter (*boxcox* lmbda parameter). Options are:

'pearsonr' (default)

Maximizes the Pearson correlation coefficient between y = boxcox(x) and the expected values for y if *x* would be normally-distributed.

'mle'

Minimizes the log-likelihood *boxcox_llf*. This is the method used in *boxcox*.

'all'

Use all optimization methods available, and return all results. Useful to compare different methods.

Returns

maxlog : float or ndarray

The optimal transform parameter found. An array instead of a scalar for method='all'.

See also:

boxcox, *boxcox_llf*, *boxcox_normplot*

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
>>> np.random.seed(1234)  # make this example reproducible
```

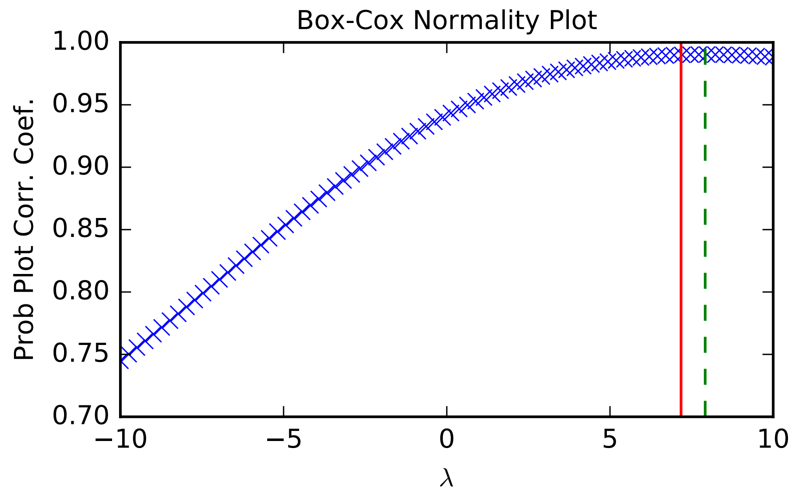
Generate some data and determine optimal lmbda in various ways:

```
>>> x = stats.loggamma.rvs(5, size=30) + 5
>>> y, lmax_mle = stats.boxcox(x)
>>> lmax_pearsonr = stats.boxcox_normmax(x)
```

```
>>> lmax_mle
7.177...
>>> lmax_pearsonr
7.916...
>>> stats.boxcox_normmax(x, method='all')
array([ 7.91667384,  7.17718692])
```

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> prob = stats.boxcox_normplot(x, -10, 10, plot=ax)
>>> ax.axvline(lmax_mle, color='r')
>>> ax.axvline(lmax_pearsonr, color='g', ls='--')
```

```
>>> plt.show()
```



`scipy.stats.`**`boxcox_llf`**(*lmb*, *data*)

The boxcox log-likelihood function.

Parameters **lmb** : scalar

Parameter for Box-Cox transformation. See `boxcox` for details.

data : array_like

Data to calculate Box-Cox log-likelihood for. If *data* is multi-dimensional, the log-likelihood is calculated along the first axis.

Returns **llf** : float or ndarray

Box-Cox log-likelihood of *data* given *lmb*. A float for 1-D *data*, an array otherwise.

See also:

`boxcox`, `probplot`, `boxcox_normplot`, `boxcox_normmax`

Notes

The Box-Cox log-likelihood function is defined here as

$$llf = (\lambda - 1) \sum_i (\log(x_i)) - N/2 \log(\sum_i (y_i - \bar{y})^2 / N),$$

where `y` is the Box-Cox transformed input data `x`.

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.axes_grid1.inset_locator import inset_axes
>>> np.random.seed(1245)
```

Generate some random variates and calculate Box-Cox log-likelihood values for them for a range of `lmbda` values:

```
>>> x = stats.loggamma.rvs(5, loc=10, size=1000)
>>> lmbdas = np.linspace(-2, 10)
>>> llf = np.zeros(lmbdas.shape, dtype=float)
>>> for ii, lmbda in enumerate(lmbdas):
...     llf[ii] = stats.boxcox_llf(lmbda, x)
```

Also find the optimal lmbda value with `boxcox`:

```
>>> x_most_normal, lmbda_optimal = stats.boxcox(x)
```

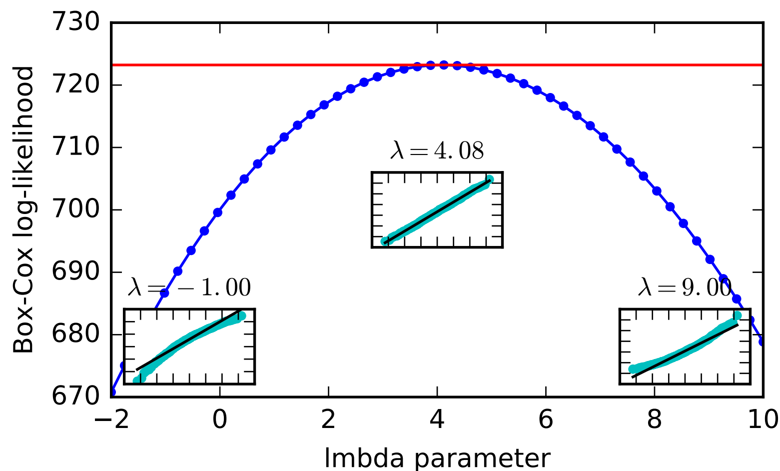
Plot the log-likelihood as function of lmbda. Add the optimal lmbda as a horizontal line to check that that's really the optimum:

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(lmbdas, llf, 'b.-')
>>> ax.axhline(stats.boxcox_llf(lmbda_optimal, x), color='r')
>>> ax.set_xlabel('lmbda parameter')
>>> ax.set_ylabel('Box-Cox log-likelihood')
```

Now add some probability plots to show that where the log-likelihood is maximized the data transformed with `boxcox` looks closest to normal:

```
>>> locs = [3, 10, 4]  # 'lower left', 'center', 'lower right'
>>> for lmbda, loc in zip([-1, lmbda_optimal, 9], locs):
...     xt = stats.boxcox(x, lmbda=lmbda)
...     (osm, osr), (slope, intercept, r_sq) = stats.probplot(xt)
...     ax_inset = inset_axes(ax, width="20%", height="20%", loc=loc)
...     ax_inset.plot(osm, osr, 'c.', osm, slope*osm + intercept, 'k-')
...     ax_inset.set_xticklabels([])
...     ax_inset.set_yticklabels([])
...     ax_inset.set_title('$\lambda=%1.2f$' % lmbda)
```

```
>>> plt.show()
```



`scipy.stats.entropy`(*pk*, *qk=None*, *base=None*)

Calculate the entropy of a distribution for given probability values.

If only probabilities *pk* are given, the entropy is calculated as `S = -sum(pk * log(pk), axis=0)`.

If *qk* is not None, then compute the Kullback-Leibler divergence `S = sum(pk * log(pk / qk), axis=0)`.

This routine will normalize *pk* and *qk* if they don't sum to 1.

Parameters

pk : sequence
: Defines the (discrete) distribution. `pk[i]` is the (possibly unnormalized) probability of event `i`.

qk : sequence, optional
: Sequence against which the relative entropy is computed. Should be in the same format as *pk*.

base : float, optional
: The logarithmic base to use, defaults to `e` (natural logarithm).

Returns

S : float
: The calculated entropy.

`chisqprob`(*args, **kwds)	`chisqprob` is deprecated!
`betai`(*args, **kwds)	`betai` is deprecated!

`scipy.stats.`**`chisqprob`**(**args*, ***kwds*)

`chisqprob` is deprecated! stats.chisqprob is deprecated in scipy 0.17.0; use stats.distributions.chi2.sf instead.

Probability value (1-tail) for the Chi^2 probability distribution.
Broadcasting rules apply.

Parameters

chisq : array_like or float > 0
: df : array_like or float, probably int >= 1

Returns

chisqprob : ndarray
: The area from *chisq* to infinity under the Chi^2 probability distribution with degrees of freedom *df*.

`scipy.stats.`**`betai`**(**args*, ***kwds*)

`betai` is deprecated! stats.betai is deprecated in scipy 0.17.0; use special.betainc instead

Returns the incomplete beta function.
I_x(a,b) = 1/B(a,b)*(Integral(0,x) of t^(a-1)(1-t)^(b-1) dt)
where a,b>0 and B(a,b) = G(a)*G(b)/(G(a+b)) where G(a) is the gamma function of a.
The standard broadcasting rules apply to a, b, and x.

Parameters

a : array_like or float > 0
: b : array_like or float > 0
: ***x*** [array_like or float] x will be clipped to be no greater than 1.0 .

Returns

betai : ndarray
: Incomplete beta function.

5.27.5 Circular statistical functions

`circmean`(samples[, high, low, axis])	Compute the circular mean for samples in a range.
`circvar`(samples[, high, low, axis])	Compute the circular variance for samples assumed to be in a range
`circstd`(samples[, high, low, axis])	Compute the circular standard deviation for samples assumed to be in the range [low to high].

`scipy.stats.`**`circmean`**(*samples*, *high=6.283185307179586*, *low=0*, *axis=None*)

Compute the circular mean for samples in a range.

Parameters

samples : array_like
: Input array.

high : float or int, optional
High boundary for circular mean range. Default is `2*pi`.

low : float or int, optional
Low boundary for circular mean range. Default is 0.

axis : int, optional
Axis along which means are computed. The default is to compute the mean of the flattened array.

Returns **circmean** : float
Circular mean.

`scipy.stats.circvar`(*samples*, *high=6.283185307179586*, *low=0*, *axis=None*)

Compute the circular variance for samples assumed to be in a range

Parameters **samples** : array_like
Input array.

low : float or int, optional
Low boundary for circular variance range. Default is 0.

high : float or int, optional
High boundary for circular variance range. Default is `2*pi`.

axis : int, optional
Axis along which variances are computed. The default is to compute the variance of the flattened array.

Returns **circvar** : float
Circular variance.

Notes

This uses a definition of circular variance that in the limit of small angles returns a number close to the 'linear' variance.

`scipy.stats.circstd`(*samples*, *high=6.283185307179586*, *low=0*, *axis=None*)

Compute the circular standard deviation for samples assumed to be in the range [low to high].

Parameters **samples** : array_like
Input array.

low : float or int, optional
Low boundary for circular standard deviation range. Default is 0.

high : float or int, optional
High boundary for circular standard deviation range. Default is `2*pi`.

axis : int, optional
Axis along which standard deviations are computed. The default is to compute the standard deviation of the flattened array.

Returns **circstd** : float
Circular standard deviation.

Notes

This uses a definition of circular standard deviation that in the limit of small angles returns a number close to the 'linear' standard deviation.

5.27.6 Contingency table functions

chi2_contingency(observed[, correction, lambda_])	Chi-square test of independence of variables in a contingency table.
contingency.expected_freq(observed)	Compute the expected frequencies from a contingency table.

Continued on next page

Table 5.288 – continued from previous page

contingency.margins(a)	Return a list of the marginal sums of the array *a*.
fisher_exact(table[, alternative])	Performs a Fisher exact test on a 2x2 contingency table.

scipy.stats.**chi2_contingency**(*observed*, *correction=True*, *lambda_=None*)

Chi-square test of independence of variables in a contingency table.

This function computes the chi-square statistic and p-value for the hypothesis test of independence of the observed frequencies in the contingency table *[R564]* *observed*. The expected frequencies are computed based on the marginal sums under the assumption of independence; see *scipy.stats.contingency.expected_freq*. The number of degrees of freedom is (expressed using numpy functions and attributes):

```
dof = observed.size - sum(observed.shape) + observed.ndim - 1
```

Parameters

observed : array_like
The contingency table. The table contains the observed frequencies (i.e. number of occurrences) in each category. In the two-dimensional case, the table is often described as an "R x C table".

correction : bool, optional
If True, *and* the degrees of freedom is 1, apply Yates' correction for continuity. The effect of the correction is to adjust each observed value by 0.5 towards the corresponding expected value.

lambda_ : float or str, optional.
By default, the statistic computed in this test is Pearson's chi-squared statistic *[R565]*. *lambda_* allows a statistic from the Cressie-Read power divergence family *[R566]* to be used instead. See *power_divergence* for details.

Returns

chi2 : float
The test statistic.

p : float
The p-value of the test

dof : int
Degrees of freedom

expected : ndarray, same shape as *observed*
The expected frequencies, based on the marginal sums of the table.

See also:

contingency.expected_freq, *fisher_exact*, *chisquare*, *power_divergence*

Notes

An often quoted guideline for the validity of this calculation is that the test should be used only if the observed and expected frequency in each cell is at least 5.

This is a test for the independence of different categories of a population. The test is only meaningful when the dimension of *observed* is two or more. Applying the test to a one-dimensional table will always result in *expected* equal to *observed* and a chi-square statistic equal to 0.

This function does not handle masked arrays, because the calculation does not make sense with missing values.

Like stats.chisquare, this function computes a chi-square statistic; the convenience this function provides is to figure out the expected frequencies and degrees of freedom from the given contingency table. If these were already known, and if the Yates' correction was not required, one could use stats.chisquare. That is, if one calls:

```
chi2, p, dof, ex = chi2_contingency(obs, correction=False)
```

then the following is true:

```
(chi2, p) == stats.chisquare(obs.ravel(), f_exp=ex.ravel(),
                             ddof=obs.size - 1 - dof)
```

The *lambda_* argument was added in version 0.13.0 of scipy.

References

[R564], [R565], [R566]

Examples

A two-way example (2 x 3):

```
>>> from scipy.stats import chi2_contingency
>>> obs = np.array([[10, 10, 20], [20, 20, 20]])
>>> chi2_contingency(obs)
(2.7777777777777777,
 0.24935220877729619,
 2,
 array([[ 12.,  12.,  16.],
        [ 18.,  18.,  24.]]))
```

Perform the test using the log-likelihood ratio (i.e. the "G-test") instead of Pearson's chi-squared statistic.

```
>>> g, p, dof, expctd = chi2_contingency(obs, lambda_="log-likelihood")
>>> g, p
(2.7688587616781319, 0.25046668010954165)
```

A four-way example (2 x 2 x 2 x 2):

```
>>> obs = np.array(
...     [[[[12, 17],
...        [11, 16]],
...       [[11, 12],
...        [15, 16]]],
...      [[[23, 15],
...        [30, 22]],
...       [[14, 17],
...        [15, 16]]]])
>>> chi2_contingency(obs)
(8.7584514426741897,
 0.64417725029295503,
 11,
 array([[[[ 14.15462386,  14.15462386],
          [ 16.49423111,  16.49423111]],
         [[ 11.2461395 ,  11.2461395 ],
          [ 13.10500554,  13.10500554]]],
        [[[ 19.5591166 ,  19.5591166 ],
          [ 22.79202844,  22.79202844]],
         [[ 15.54012004,  15.54012004],
          [ 18.10873492,  18.10873492]]]]))
```

scipy.stats.contingency.**expected_freq**(*observed*)

Compute the expected frequencies from a contingency table.

Given an n-dimensional contingency table of observed frequencies, compute the expected frequencies for the table based on the marginal sums under the assumption that the groups associated with each dimension are independent.

Parameters **observed** : array_like
The table of observed frequencies. (While this function can handle a 1-D array, that case is trivial. Generally *observed* is at least 2-D.)

Returns **expected** : ndarray of float64
The expected frequencies, based on the marginal sums of the table. Same shape as *observed*.

Examples

```
>>> observed = np.array([[10, 10, 20],[20, 20, 20]])
>>> from scipy.stats import expected_freq
>>> expected_freq(observed)
array([[ 12.,  12.,  16.],
       [ 18.,  18.,  24.]])
```

`scipy.stats.contingency.margins(a)`
Return a list of the marginal sums of the array *a*.

Parameters **a** : ndarray
The array for which to compute the marginal sums.

Returns **margsums** : list of ndarrays
A list of length *a.ndim*. *margsums[k]* is the result of summing *a* over all axes except *k*; it has the same number of dimensions as *a*, but the length of each axis except axis *k* will be 1.

Examples

```
>>> a = np.arange(12).reshape(2, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> m0, m1 = margins(a)
>>> m0
array([[15],
       [51]])
>>> m1
array([[ 6,  8, 10, 12, 14, 16]])
```

```
>>> b = np.arange(24).reshape(2,3,4)
>>> m0, m1, m2 = margins(b)
>>> m0
array([[[ 66]],
       [[210]]])
>>> m1
array([[[ 60],
        [ 92],
        [124]]])
>>> m2
array([[[60, 66, 72, 78]]])
```

`scipy.stats.fisher_exact(table, alternative='two-sided')`
Performs a Fisher exact test on a 2x2 contingency table.

Parameters **table** : array_like of ints
A 2x2 contingency table. Elements should be non-negative integers.

alternative : {'two-sided', 'less', 'greater'}, optional
Which alternative hypothesis to the null hypothesis the test uses. Default is 'two-sided'.

Returns **oddsratio** : float

This is prior odds ratio and not a posterior estimate.

p_value : float

P-value, the probability of obtaining a distribution at least as extreme as the one that was actually observed, assuming that the null hypothesis is true.

See also:

chi2_contingency
Chi-square test of independence of variables in a contingency table.

Notes

The calculated odds ratio is different from the one R uses. This scipy implementation returns the (more common) "unconditional Maximum Likelihood Estimate", while R uses the "conditional Maximum Likelihood Estimate".

For tables with large numbers, the (inexact) chi-square test implemented in the function *chi2_contingency* can also be used.

Examples

Say we spend a few days counting whales and sharks in the Atlantic and Indian oceans. In the Atlantic ocean we find 8 whales and 1 shark, in the Indian ocean 2 whales and 5 sharks. Then our contingency table is:

```
        Atlantic  Indian
whales     8         2
sharks     1         5
```

We use this table to find the p-value:

```
>>> import scipy.stats as stats
>>> oddsratio, pvalue = stats.fisher_exact([[8, 2], [1, 5]])
>>> pvalue
0.0349...
```

The probability that we would observe this or an even more imbalanced ratio by chance is about 3.5%. A commonly used significance level is 5%–if we adopt that, we can therefore conclude that our observed imbalance is statistically significant; whales prefer the Atlantic while sharks prefer the Indian ocean.

5.27.7 Plot-tests

ppcc_max(x[, brack, dist])	Calculate the shape parameter that maximizes the PPCC
ppcc_plot(x, a, b[, dist, plot, N])	Calculate and optionally plot probability plot correlation coefficient.
probplot(x[, sparams, dist, fit, plot, rvalue])	Calculate quantiles for a probability plot, and optionally show the plot.
boxcox_normplot(x, la, lb[, plot, N])	Compute parameters for a Box-Cox normality plot, optionally show it.

scipy.stats.**ppcc_max**(*x*, *brack=(0.0, 1.0)*, *dist='tukeylambda'*)
Calculate the shape parameter that maximizes the PPCC

The probability plot correlation coefficient (PPCC) plot can be used to determine the optimal shape parameter for a one-parameter family of distributions. ppcc_max returns the shape parameter that would maximize the probability plot correlation coefficient for the given data to a one-parameter family of distributions.

Parameters **x** : array_like

Input array.

brack : tuple, optional
Triple (a,b,c) where (a<b<c). If bracket consists of two numbers (a, c) then they are assumed to be a starting interval for a downhill bracket search (see *scipy.optimize.brent*).

dist : str or stats.distributions instance, optional
Distribution or distribution function name. Objects that look enough like a stats.distributions instance (i.e. they have a `ppf` method) are also accepted. The default is `'tukeylambda'`.

Returns **shape_value** : float
The shape parameter at which the probability plot correlation coefficient reaches its max value.

See also:

ppcc_plot, *probplot*, *boxcox*

Notes

The brack keyword serves as a starting point which is useful in corner cases. One can use a plot to obtain a rough visual estimate of the location for the maximum to start the search near it.

References

[R626], [R627]

Examples

First we generate some random data from a Tukey-Lambda distribution, with shape parameter -0.7:

```
>>> from scipy import stats
>>> x = stats.tukeylambda.rvs(-0.7, loc=2, scale=0.5, size=10000,
...                           random_state=1234567) + 1e4
```

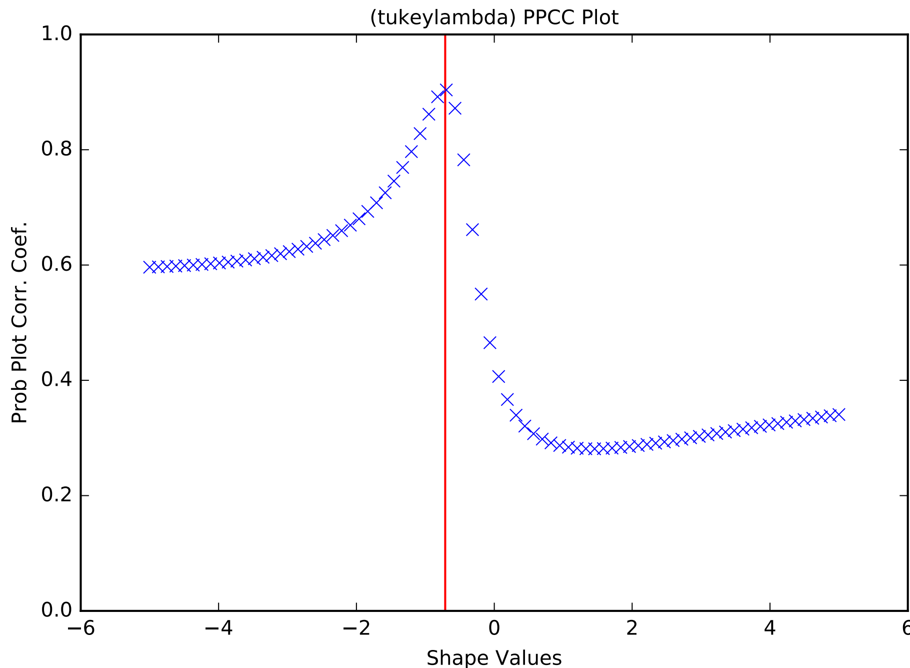
Now we explore this data with a PPCC plot as well as the related probability plot and Box-Cox normplot. A red line is drawn where we expect the PPCC value to be maximal (at the shape parameter -0.7 used above):

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure(figsize=(8, 6))
>>> ax = fig.add_subplot(111)
>>> res = stats.ppcc_plot(x, -5, 5, plot=ax)
```

We calculate the value where the shape should reach its maximum and a red line is drawn there. The line should coincide with the highest point in the ppcc_plot.

```
>>> max = stats.ppcc_max(x)
>>> ax.vlines(max, 0, 1, colors='r', label='Expected shape value')
```

```
>>> plt.show()
```



scipy.stats.**ppcc_plot** (*x*, *a*, *b*, *dist='tukeylambda'*, *plot=None*, *N=80*)

Calculate and optionally plot probability plot correlation coefficient.

The probability plot correlation coefficient (PPCC) plot can be used to determine the optimal shape parameter for a one-parameter family of distributions. It cannot be used for distributions without shape parameters (like the normal distribution) or with multiple shape parameters.

By default a Tukey-Lambda distribution (*stats.tukeylambda*) is used. A Tukey-Lambda PPCC plot interpolates from long-tailed to short-tailed distributions via an approximately normal one, and is therefore particularly useful in practice.

Parameters

x : array_like
: Input array.

a, b: scalar
: Lower and upper bounds of the shape parameter to use.

dist : str or stats.distributions instance, optional
: Distribution or distribution function name. Objects that look enough like a stats.distributions instance (i.e. they have a `ppf` method) are also accepted. The default is `'tukeylambda'`.

plot : object, optional
: If given, plots PPCC against the shape parameter. *plot* is an object that has to have methods "plot" and "text". The `matplotlib.pyplot` module or a Matplotlib Axes object can be used, or a custom object with the same methods. Default is None, which means that no plot is created.

N : int, optional
: Number of points on the horizontal axis (equally distributed from *a* to *b*).

Returns

svals : ndarray

The shape values for which *ppcc* was calculated.

ppcc : ndarray

The calculated probability plot correlation coefficient values.

See also:

ppcc_max, *probplot*, *boxcox_normplot*, *tukeylambda*

References

J.J. Filliben, "The Probability Plot Correlation Coefficient Test for Normality", Technometrics, Vol. 17, pp. 111-117, 1975.

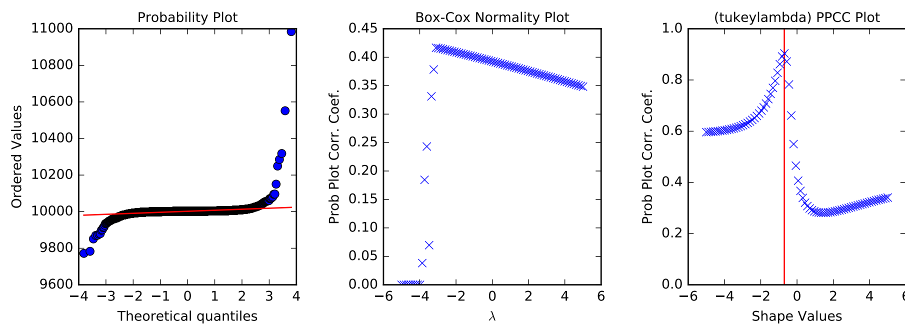
Examples

First we generate some random data from a Tukey-Lambda distribution, with shape parameter -0.7:

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
>>> np.random.seed(1234567)
>>> x = stats.tukeylambda.rvs(-0.7, loc=2, scale=0.5, size=10000) + 1e4
```

Now we explore this data with a PPCC plot as well as the related probability plot and Box-Cox normplot. A red line is drawn where we expect the PPCC value to be maximal (at the shape parameter -0.7 used above):

```
>>> fig = plt.figure(figsize=(12, 4))
>>> ax1 = fig.add_subplot(131)
>>> ax2 = fig.add_subplot(132)
>>> ax3 = fig.add_subplot(133)
>>> res = stats.probplot(x, plot=ax1)
>>> res = stats.boxcox_normplot(x, -5, 5, plot=ax2)
>>> res = stats.ppcc_plot(x, -5, 5, plot=ax3)
>>> ax3.vlines(-0.7, 0, 1, colors='r', label='Expected shape value')
>>> plt.show()
```



`scipy.stats.`**`probplot`**(*x*, *sparams=()*, *dist='norm'*, *fit=True*, *plot=None*, *rvalue=False*)

Calculate quantiles for a probability plot, and optionally show the plot.

Generates a probability plot of sample data against the quantiles of a specified theoretical distribution (the normal distribution by default). `probplot` optionally calculates a best-fit line for the data and plots the results using Matplotlib or a given plot function.

Parameters **x** : array_like

Sample/response data from which `probplot` creates the plot.

sparams : tuple, optional

Distribution-specific shape parameters (shape parameters plus location and scale).

dist : str or stats.distributions instance, optional
Distribution or distribution function name. The default is 'norm' for a normal probability plot. Objects that look enough like a stats.distributions instance (i.e. they have a `ppf` method) are also accepted.

fit : bool, optional
Fit a least-squares regression (best-fit) line to the sample data if True (default).

plot : object, optional
If given, plots the quantiles and least squares fit. *plot* is an object that has to have methods "plot" and "text". The `matplotlib.pyplot` module or a Matplotlib Axes object can be used, or a custom object with the same methods. Default is None, which means that no plot is created.

Returns

(osm, osr) : tuple of ndarrays
Tuple of theoretical quantiles (osm, or order statistic medians) and ordered responses (osr). *osr* is simply sorted input *x*. For details on how *osm* is calculated see the Notes section.

(slope, intercept, r) : tuple of floats, optional
Tuple containing the result of the least-squares fit, if that is performed by `probplot`. *r* is the square root of the coefficient of determination. If `fit=False` and `plot=None`, this tuple is not returned.

Notes

Even if *plot* is given, the figure is not shown or saved by `probplot`; `plt.show()` or `plt.savefig('figname.png')` should be used after calling `probplot`.

`probplot` generates a probability plot, which should not be confused with a Q-Q or a P-P plot. Statsmodels has more extensive functionality of this type, see `statsmodels.api.ProbPlot`.

The formula used for the theoretical quantiles (horizontal axis of the probability plot) is Filliben's estimate:

```
quantiles = dist.ppf(val), for

        0.5**(1/n),                  for i = n
  val = (i - 0.3175) / (n + 0.365),  for i = 2, ..., n-1
        1 - 0.5**(1/n),              for i = 1
```

where `i` indicates the i-th ordered value and `n` is the total number of values.

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
>>> nsample = 100
>>> np.random.seed(7654321)
```

A t distribution with small degrees of freedom:

```
>>> ax1 = plt.subplot(221)
>>> x = stats.t.rvs(3, size=nsample)
>>> res = stats.probplot(x, plot=plt)
```

A t distribution with larger degrees of freedom:

```
>>> ax2 = plt.subplot(222)
>>> x = stats.t.rvs(25, size=nsample)
>>> res = stats.probplot(x, plot=plt)
```

A mixture of two normal distributions with broadcasting:

```
>>> ax3 = plt.subplot(223)
>>> x = stats.norm.rvs(loc=[0,5], scale=[1,1.5],
...                    size=(nsample//2,2)).ravel()
>>> res = stats.probplot(x, plot=plt)
```

A standard normal distribution:

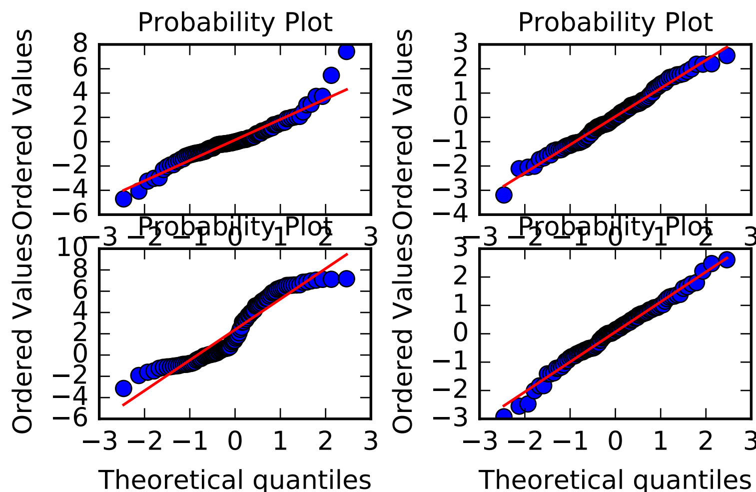
```
>>> ax4 = plt.subplot(224)
>>> x = stats.norm.rvs(loc=0, scale=1, size=nsample)
>>> res = stats.probplot(x, plot=plt)
```

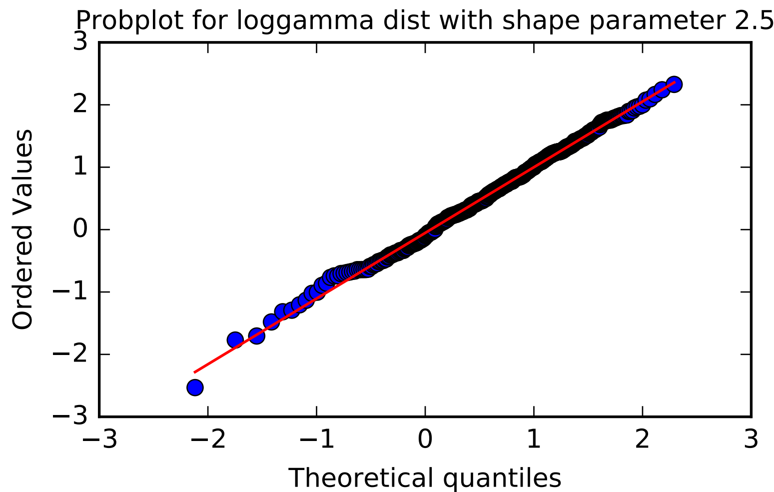
Produce a new figure with a loggamma distribution, using the `dist` and `sparams` keywords:

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> x = stats.loggamma.rvs(c=2.5, size=500)
>>> res = stats.probplot(x, dist=stats.loggamma, sparams=(2.5,), plot=ax)
>>> ax.set_title("Probplot for loggamma dist with shape parameter 2.5")
```

Show the results with Matplotlib:

```
>>> plt.show()
```





`scipy.stats.boxcox_normplot` (*x*, *la*, *lb*, *plot=None*, *N=80*)

Compute parameters for a Box-Cox normality plot, optionally show it.

A Box-Cox normality plot shows graphically what the best transformation parameter is to use in *boxcox* to obtain a distribution that is close to normal.

Parameters

x : array_like
: Input array.

la, lb : scalar
: The lower and upper bounds for the `lmbda` values to pass to *boxcox* for Box-Cox transformations. These are also the limits of the horizontal axis of the plot if that is generated.

plot : object, optional
: If given, plots the quantiles and least squares fit. *plot* is an object that has to have methods "plot" and "text". The `matplotlib.pyplot` module or a Matplotlib Axes object can be used, or a custom object with the same methods. Default is None, which means that no plot is created.

N : int, optional
: Number of points on the horizontal axis (equally distributed from *la* to *lb*).

Returns

lmbdas : ndarray
: The `lmbda` values for which a Box-Cox transform was done.

ppcc : ndarray
: Probability Plot Correlelation Coefficient, as obtained from *probplot* when fitting the Box-Cox transformed input *x* against a normal distribution.

See also:

probplot, *boxcox*, *boxcox_normmax*, *boxcox_llf*, *ppcc_max*

Notes

Even if *plot* is given, the figure is not shown or saved by *boxcox_normplot*; `plt.show()` or `plt.savefig('figname.png')` should be used after calling *probplot*.

Examples

```
>>> from scipy import stats
>>> import matplotlib.pyplot as plt
```

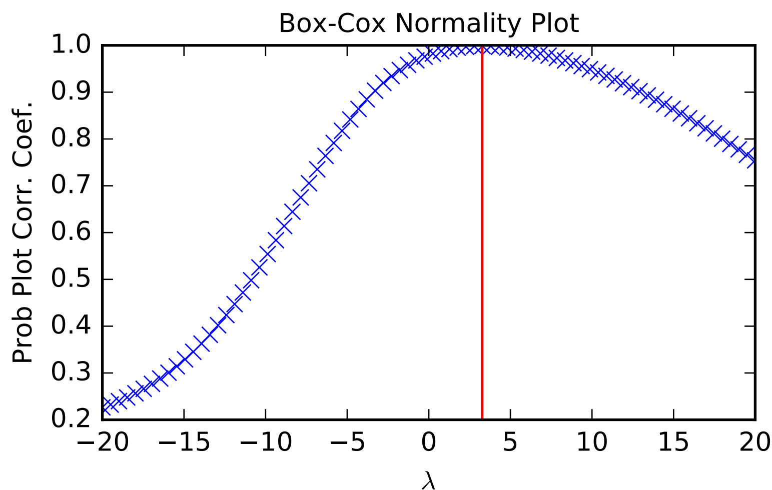
Generate some non-normally distributed data, and create a Box-Cox plot:

```
>>> x = stats.loggamma.rvs(5, size=500) + 5
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> prob = stats.boxcox_normplot(x, -20, 20, plot=ax)
```

Determine and plot the optimal `lmbda` to transform `x` and plot it in the same plot:

```
>>> _, maxlog = stats.boxcox(x)
>>> ax.axvline(maxlog, color='r')
```

```
>>> plt.show()
```



5.27.8 Masked statistics functions

Statistical functions for masked arrays (`scipy.stats.mstats`)

This module contains a large number of statistical functions that can be used with masked arrays.

Most of these functions are similar to those in scipy.stats but might have small differences in the API or in the algorithm used. Since this is a relatively new package, some API changes are still possible.

argstoarray(*args)	Constructs a 2D array from a group of sequences.
betai(*args, **kwds)	*betai* is deprecated!
chisquare(f_obs[, f_exp, ddof, axis])	Calculates a one-way chi square test.
count_tied_groups(x[, use_missing])	Counts the number of tied values.
describe(a[, axis, ddof, bias])	Computes several descriptive statistics of the passed array.
f_oneway(*args)	Performs a 1-way ANOVA, returning an F-value and probability given any number of groups.
f_value_wilks_lambda(*args, **kwds)	*f_value_wilks_lambda* is deprecated!
find_repeats(arr)	Find repeats in arr and return a tuple (repeats, repeat_count).

Continued on next page

Table 5.290 – continued from previous page

friedmanchisquare(*args)	Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA.
kendalltau(x, y[, use_ties, use_missing])	Computes Kendall's rank correlation tau on two variables *x* and *y*.
kendalltau_seasonal(x)	Computes a multivariate Kendall's rank correlation tau, for seasonal data.
kruskalwallis(*args)	Compute the Kruskal-Wallis H-test for independent samples
ks_twosamp(data1, data2[, alternative])	Computes the Kolmogorov-Smirnov test on two samples.
kurtosis(a[, axis, fisher, bias])	Computes the kurtosis (Fisher or Pearson) of a dataset.
kurtosistest(a[, axis])	Tests whether a dataset has normal kurtosis
linregress(x[, y])	Calculate a linear least-squares regression for two sets of measurements.
mannwhitneyu(x, y[, use_continuity])	Computes the Mann-Whitney statistic
plotting_positions(data[, alpha, beta])	Returns plotting positions (or empirical percentile points) for the data.
mode(a[, axis])	Returns an array of the modal (most common) value in the passed array.
moment(a[, moment, axis])	Calculates the nth moment about the mean for a sample.
mquantiles(a[, prob, alphap, betap, axis, limit])	Computes empirical quantiles for a data array.
msign(x)	Returns the sign of x, or 0 if x is masked.
normaltest(a[, axis])	Tests whether a sample differs from a normal distribution.
obrientransform(*args)	Computes a transform on input data (any number of columns).
pearsonr(x, y)	Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.
plotting_positions(data[, alpha, beta])	Returns plotting positions (or empirical percentile points) for the data.
pointbiserialr(x, y)	Calculates a point biserial correlation coefficient and its p-value.
rankdata(data[, axis, use_missing])	Returns the rank (also known as order statistics) of each data point along the given axis.
scoreatpercentile(data, per[, limit, ...])	Calculate the score at the given 'per' percentile of the sequence a.
sem(a[, axis, ddof])	Calculates the standard error of the mean of the input array.
signaltonoise(*args, **kwds)	*signaltonoise* is deprecated!
skew(a[, axis, bias])	Computes the skewness of a data set.
skewtest(a[, axis])	Tests whether the skew is different from the normal distribution.
spearmanr(x, y[, use_ties])	Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.
theilslopes(y[, x, alpha])	Computes the Theil-Sen estimator for a set of points (x, y).
threshold(*args, **kwds)	*threshold* is deprecated!
tmax(a[, upperlimit, axis, inclusive])	Compute the trimmed maximum
tmean(a[, limits, inclusive, axis])	Compute the trimmed mean.
tmin(a[, lowerlimit, axis, inclusive])	Compute the trimmed minimum
trim(a[, limits, inclusive, relative, axis])	Trims an array by masking the data outside some given limits.
trima(a[, limits, inclusive])	Trims an array by masking the data outside some given limits.

Continued on next page

Table 5.290 – continued from previous page

trimboth(data[, proportiontocut, inclusive, ...])	Trims the smallest and largest data values.
trimmed_stde(a[, limits, inclusive, axis])	Returns the standard error of the trimmed mean along the given axis.
trimr(a[, limits, inclusive, axis])	Trims an array by masking some proportion of the data on each end.
trimtail(data[, proportiontocut, tail, ...])	Trims the data by masking values from one tail.
tsem(a[, limits, inclusive, axis, ddof])	Compute the trimmed standard error of the mean.
ttest_onesamp(a, popmean[, axis])	Calculates the T-test for the mean of ONE group of scores.
ttest_ind(a, b[, axis, equal_var])	Calculates the T-test for the means of TWO INDEPENDENT samples of scores.
ttest_onesamp(a, popmean[, axis])	Calculates the T-test for the mean of ONE group of scores.
ttest_rel(a, b[, axis])	Calculates the T-test on TWO RELATED samples of scores, a and b.
tvar(a[, limits, inclusive, axis, ddof])	Compute the trimmed variance
variation(a[, axis])	Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.
winsorize(a[, limits, inclusive, inplace, axis])	Returns a Winsorized version of the input array.
zmap(scores, compare[, axis, ddof])	Calculates the relative z-scores.
zscore(a[, axis, ddof])	Calculates the z score of each value in the sample, relative to the sample mean and standard deviation.
compare_medians_ms(group_1, group_2[, axis])	Compares the medians from two independent groups along the given axis.
gmean(a[, axis, dtype])	Compute the geometric mean along the specified axis.
hdmedian(data[, axis, var])	Returns the Harrell-Davis estimate of the median along the given axis.
hdquantiles(data[, prob, axis, var])	Computes quantile estimates with the Harrell-Davis method.
hdquantiles_sd(data[, prob, axis])	The standard error of the Harrell-Davis quantile estimates by jackknife.
hmean(a[, axis, dtype])	Calculates the harmonic mean along the specified axis.
idealfourths(data[, axis])	Returns an estimate of the lower and upper quartiles.
kruskal(*args)	Compute the Kruskal-Wallis H-test for independent samples
ks_2samp(data1, data2[, alternative])	Computes the Kolmogorov-Smirnov test on two samples.
median_cihs(data[, alpha, axis])	Computes the alpha-level confidence interval for the median of the data.
meppf(data[, alpha, beta])	Returns plotting positions (or empirical percentile points) for the data.
mjci(data[, prob, axis])	Returns the Maritz-Jarrett estimators of the standard error of selected experimental quantiles of the data.
mquantiles_cimj(data[, prob, alpha, axis])	Computes the alpha confidence interval for the selected quantiles of the data, with Maritz-Jarrett estimators.
rsh(data[, points])	Evaluates Rosenblatt's shifted histogram estimators for each point on the dataset 'data'.
sen_seasonal_slopes(x)	
trimmed_mean(a[, limits, inclusive, ...])	
trimmed_mean_ci(data[, limits, inclusive, ...])	Selected confidence interval of the trimmed mean along the given axis.
trimmed_std(a[, limits, inclusive, ...])	
trimmed_var(a[, limits, inclusive, ...])	

Continued on next page

Table 5.290 – continued from previous page

ttest_1samp(a, popmean[, axis])	Calculates the T-test for the mean of ONE group of scores.

scipy.stats.mstats.**argstoarray**(**args*)

Constructs a 2D array from a group of sequences.

Sequences are filled with missing values to match the length of the longest sequence.

Parameters **args** : sequences

Group of sequences.

Returns **argstoarray** : MaskedArray

A (*m* x *n*) masked array, where *m* is the number of arguments and *n* the length of the longest argument.

Notes

numpy.ma.row_stack has identical behavior, but is called with a sequence of sequences.

scipy.stats.mstats.**betai**(**args*, ***kwds*)

betai is deprecated! mstats.betai is deprecated in scipy 0.17.0; use special.betainc instead.

betai() is deprecated in scipy 0.17.0.
For details about this function, see *stats.betai*.

scipy.stats.mstats.**chisquare**(*f_obs*, *f_exp=None*, *ddof=0*, *axis=0*)

Calculates a one-way chi square test.

The chi square test tests the null hypothesis that the categorical data has the given frequencies.

Parameters **f_obs** : array_like

Observed frequencies in each category.

f_exp : array_like, optional

Expected frequencies in each category. By default the categories are assumed to be equally likely.

ddof : int, optional

"Delta degrees of freedom": adjustment to the degrees of freedom for the p-value. The p-value is computed using a chi-squared distribution with `k - 1 - ddof` degrees of freedom, where *k* is the number of observed frequencies. The default value of *ddof* is 0.

axis : int or None, optional

The axis of the broadcast result of *f_obs* and *f_exp* along which to apply the test. If axis is None, all values in *f_obs* are treated as a single data set. Default is 0.

Returns **chisq** : float or ndarray

The chi-squared test statistic. The value is a float if *axis* is None or *f_obs* and *f_exp* are 1-D.

p : float or ndarray

The p-value of the test. The value is a float if *ddof* and the return value *chisq* are scalars.

See also:

power_divergence, mstats.chisquare

Notes

This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5.

The default degrees of freedom, k-1, are for the case when no parameters of the distribution are estimated. If p parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are k-1-p. If

the parameters are estimated in a different way, then the dof can be between k-1-p and k-1. However, it is also possible that the asymptotic distribution is not a chisquare, in which case this test is not appropriate.

References

[R610], *[R611]*

Examples

When just *f_obs* is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies.

```
>>> from scipy.stats import chisquare
>>> chisquare([16, 18, 16, 14, 12, 12])
(2.0, 0.84914503608460956)
```

With *f_exp* the expected frequencies can be given.

```
>>> chisquare([16, 18, 16, 14, 12, 12], f_exp=[16, 16, 16, 16, 16, 8])
(3.5, 0.62338762774958223)
```

When *f_obs* is 2-D, by default the test is applied to each column.

```
>>> obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20, 24]]).T
>>> obs.shape
(6, 2)
>>> chisquare(obs)
(array([ 2.        ,  6.66666667]), array([ 0.84914504,  0.24663415]))
```

By setting `axis=None`, the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```
>>> chisquare(obs, axis=None)
(23.31034482758621, 0.015975692534127565)
>>> chisquare(obs.ravel())
(23.31034482758621, 0.015975692534127565)
```

ddof is the change to make to the default degrees of freedom.

```
>>> chisquare([16, 18, 16, 14, 12, 12], ddof=1)
(2.0, 0.73575888234288467)
```

The calculation of the p-values is done by broadcasting the chi-squared statistic with *ddof*.

```
>>> chisquare([16, 18, 16, 14, 12, 12], ddof=[0,1,2])
(2.0, array([ 0.84914504,  0.73575888,  0.5724067 ]))
```

f_obs and *f_exp* are also broadcast. In the following, *f_obs* has shape (6,) and *f_exp* has shape (2, 6), so the result of broadcasting *f_obs* and *f_exp* has shape (2, 6). To compute the desired chi-squared statistics, we use `axis=1`:

```
>>> chisquare([16, 18, 16, 14, 12, 12],
...           f_exp=[[16, 16, 16, 16, 16, 8], [8, 20, 20, 16, 12, 12]],
...           axis=1)
(array([ 3.5 ,  9.25]), array([ 0.62338763,  0.09949846]))
```

`scipy.stats.mstats.`**`count_tied_groups`**(*x*, *use_missing=False*)

Counts the number of tied values.

Parameters **x** : sequence

Sequence of data on which to counts the ties

use_missing : bool, optional

Whether to consider missing values as tied.

Returns **count_tied_groups** : dict

Returns a dictionary (nb of ties: nb of groups).

Examples

```
>>> from scipy.stats import mstats
>>> z = [0, 0, 0, 2, 2, 2, 3, 3, 4, 5, 6]
>>> mstats.count_tied_groups(z)
{2: 1, 3: 2}
```

In the above example, the ties were 0 (3x), 2 (3x) and 3 (2x).

```
>>> z = np.ma.array([0, 0, 1, 2, 2, 2, 3, 3, 4, 5, 6])
>>> mstats.count_tied_groups(z)
{2: 2, 3: 1}
>>> z[[1,-1]] = np.ma.masked
>>> mstats.count_tied_groups(z, use_missing=True)
{2: 2, 3: 1}
```

`scipy.stats.mstats.`**`describe`**(*a*, *axis=0*, *ddof=0*, *bias=True*)

Computes several descriptive statistics of the passed array.

Parameters **a** : array_like

Data array

axis : int or None, optional

Axis along which to calculate statistics. Default 0. If None, compute over the whole array *a*.

ddof : int, optional

degree of freedom (default 0); note that default ddof is different from the same routine in stats.describe

bias : bool, optional

If False, then the skewness and kurtosis calculations are corrected for statistical bias.

Returns **nobs** : int

(size of the data (discarding missing values)

minmax : (int, int)

min, max

mean : float

arithmetic mean

variance : float

unbiased variance

skewness : float

biased skewness

kurtosis : float

biased kurtosis

Examples

```
>>> from scipy.stats.mstats import describe
>>> ma = np.ma.array(range(6), mask=[0, 0, 0, 1, 1, 1])
>>> describe(ma)
DescribeResult(nobs=array(3), minmax=(masked_array(data = 0,
             mask = False,
```

```
       fill_value = 999999)
, masked_array(data = 2,
             mask = False,
       fill_value = 999999)
), mean=1.0, variance=0.66666666666666663, skewness=masked_array(data = 0.0,
             mask = False,
       fill_value = 1e+20)
, kurtosis=-1.5)
```

`scipy.stats.mstats.`**`f_oneway`**`(`**args*`)`

Performs a 1-way ANOVA, returning an F-value and probability given any number of groups. From Heiman, pp.394-7.

Usage: `f_oneway(*args)`, where `*args` is 2 or more arrays, one per treatment group.

Returns **statistic** : float

The computed F-value of the test.

pvalue : float

The associated p-value from the F-distribution.

`scipy.stats.mstats.`**`f_value_wilks_lambda`**`(`**args*, ***kwds*`)`

`f_value_wilks_lambda` is deprecated! mstats.f_value_wilks_lambda deprecated in scipy 0.17.0

Calculation of Wilks lambda F-statistic for multivariate data, per

Maxwell & Delaney p.657.

`scipy.stats.mstats.`**`find_repeats`**`(`*arr*`)`

Find repeats in arr and return a tuple (repeats, repeat_count).

The input is cast to float64. Masked values are discarded.

Parameters **arr** : sequence

Input array. The array is flattened if it is not 1D.

Returns **repeats** : ndarray

Array of repeated values.

counts : ndarray

Array of counts.

`scipy.stats.mstats.`**`friedmanchisquare`**`(`**args*`)`

Friedman Chi-Square is a non-parametric, one-way within-subjects ANOVA. This function calculates the Friedman Chi-square test for repeated measures and returns the result, along with the associated probability value.

Each input is considered a given group. Ideally, the number of treatments among each group should be equal. If this is not the case, only the first n treatments are taken into account, where n is the number of treatments of the smallest group. If a group has some missing values, the corresponding treatments are masked in the other groups. The test statistic is corrected for ties.

Masked values in one group are propagated to the other groups.

Returns **statistic** : float

the test statistic.

pvalue : float

the associated p-value.

`scipy.stats.mstats.`**`kendalltau`**`(`*x*, *y*, *use_ties=True*, *use_missing=False*`)`

Computes Kendall's rank correlation tau on two variables *x* and *y*.

Parameters **x** : sequence

First data list (for example, time).

y : sequence

Second data list.

use_ties : {True, False}, optional
Whether ties correction should be performed.

use_missing : {False, True}, optional
Whether missing data should be allocated a rank of 0 (False) or the average rank (True)

Returns **correlation** : float
Kendall tau

pvalue : float
Approximate 2-side p-value.

`scipy.stats.mstats.`**`kendalltau_seasonal`**(*x*)

Computes a multivariate Kendall's rank correlation tau, for seasonal data.

Parameters **x** : 2-D ndarray
Array of seasonal data, with seasons in columns.

`scipy.stats.mstats.`**`kruskalwallis`**(**args*)

Compute the Kruskal-Wallis H-test for independent samples

Parameters **sample1, sample2, ...** : array_like
Two or more arrays with the sample measurements can be given as arguments.

Returns **statistic** : float
The Kruskal-Wallis H statistic, corrected for ties

pvalue : float
The p-value for the test using the assumption that H has a chi square distribution

Notes

For more details on `kruskal`, see *stats.kruskal*.

`scipy.stats.mstats.`**`ks_twosamp`**(*data1*, *data2*, *alternative='two-sided'*)

Computes the Kolmogorov-Smirnov test on two samples.

Missing values are discarded.

Parameters **data1** : array_like
First data set

data2 : array_like
Second data set

alternative : {'two-sided', 'less', 'greater'}, optional
Indicates the alternative hypothesis. Default is 'two-sided'.

Returns **d** : float
Value of the Kolmogorov Smirnov test

p : float
Corresponding p-value.

`scipy.stats.mstats.`**`kurtosis`**(*a*, *axis=0*, *fisher=True*, *bias=True*)

Computes the kurtosis (Fisher or Pearson) of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution.

If bias is False then the kurtosis is calculated using k statistics to eliminate bias coming from biased moment estimators

Use `kurtosistest` to see if result is close enough to normal.

Parameters **a** : array
data for which the kurtosis is calculated

axis : int or None, optional
Axis along which the kurtosis is calculated. Default is 0. If None, compute over the whole array *a*.

fisher : bool, optional
If True, Fisher's definition is used (normal ==> 0.0). If False, Pearson's definition is used (normal ==> 3.0).

bias : bool, optional
If False, then the calculations are corrected for statistical bias.

Returns **kurtosis** : array
The kurtosis of values along an axis. If all values are equal, return -3 for Fisher's definition and 0 for Pearson's definition.

Notes

For more details about `kurtosis`, see *stats.kurtosis*.

`scipy.stats.mstats.`**`kurtosistest`**(*a*, *axis=0*)
Tests whether a dataset has normal kurtosis

Parameters **a** : array
array of the sample data

axis : int or None, optional
Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

Returns **statistic** : float
The computed z-score for this test.

pvalue : float
The 2-sided p-value for the hypothesis test

Notes

For more details about `kurtosistest`, see *stats.kurtosistest*.

`scipy.stats.mstats.`**`linregress`**(*x*, *y=None*)
Calculate a linear least-squares regression for two sets of measurements.

Parameters **x, y** : array_like
Two sets of measurements. Both arrays should have the same length. If only x is given (and y=None), then it must be a two-dimensional array where one dimension has length 2. The two sets of measurements are then found by splitting the array along the length-2 dimension.

Returns **slope** : float
slope of the regression line

intercept : float
intercept of the regression line

rvalue : float
correlation coefficient

pvalue : float
two-sided p-value for a hypothesis test whose null hypothesis is that the slope is zero.

stderr : float
Standard error of the estimated gradient.

See also:

`scipy.optimize.curve_fit`
Use non-linear least squares to fit a function to data.

`scipy.optimize.leastsq`
Minimize the sum of squares of a set of equations.

Notes

Missing values are considered pair-wise: if a value is missing in x, the corresponding value in y is masked.

Examples

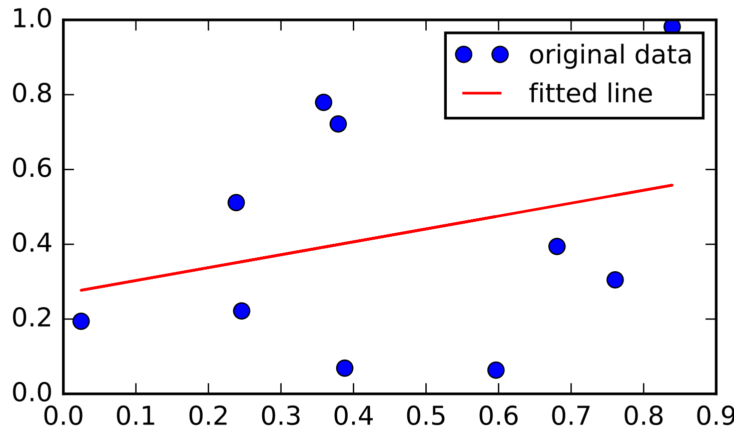
```
>>> import matplotlib.pyplot as plt
>>> from scipy import stats
>>> np.random.seed(12345678)
>>> x = np.random.random(10)
>>> y = np.random.random(10)
>>> slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
```

To get coefficient of determination (r_squared)

```
>>> print("r-squared:", r_value**2)
('r-squared:', 0.080402268539028335)
```

Plot the data along with the fitted line

```
>>> plt.plot(x, y, 'o', label='original data')
>>> plt.plot(x, intercept + slope*x, 'r', label='fitted line')
>>> plt.legend()
>>> plt.show()
```



`scipy.stats.mstats.`**`mannwhitneyu`**(*x*, *y*, *use_continuity=True*)

Computes the Mann-Whitney statistic

Missing values in *x* and/or *y* are discarded.

Parameters **x** : sequence

Input

y : sequence

Input

use_continuity : {True, False}, optional

Whether a continuity correction (1/2.) should be taken into account.

Returns **statistic** : float

The Mann-Whitney statistics

pvalue : float

Approximate p-value assuming a normal distribution.

`scipy.stats.mstats.`**`plotting_positions`**(*data*, *alpha=0.4*, *beta=0.4*)

Returns plotting positions (or empirical percentile points) for the data.

Plotting positions are defined as **`(i-alpha)/(n+1-alpha-beta)`*****, where:***

- i is the rank order statistics
- n is the number of unmasked values along the given axis
- *alpha* and *beta* are two parameters.

Typical values for* alpha *and* beta *are:

- (0,1) : `p(k) = k/n`, linear interpolation of cdf (R, type 4)
- (.5,.5) : `p(k) = (k-1/2.)/n`, piecewise linear function (R, type 5)
- (0,0) : `p(k) = k/(n+1)`, Weibull (R type 6)
- (1,1) : `p(k) = (k-1)/(n-1)`, in this case, `p(k) = mode[F(x[k])]`. That's R default (R type 7)
- (1/3,1/3): `p(k) = (k-1/3)/(n+1/3)`, then `p(k) ~ median[F(x[k])]`. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (R type 8)
- (3/8,3/8): `p(k) = (k-3/8)/(n+1/4)`, Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (R type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM
- (.3175, .3175): used in scipy.stats.probplot

Parameters

data : array_like
: Input data, as a sequence or array of dimension at most 2.

alpha : float, optional
: Plotting positions parameter. Default is 0.4.

beta : float, optional
: Plotting positions parameter. Default is 0.4.

Returns

positions : MaskedArray
: The calculated plotting positions.

`scipy.stats.mstats.`**`mode`**(*a*, *axis=0*)

Returns an array of the modal (most common) value in the passed array.

Parameters

a : array_like
: n-dimensional array of which to find mode(s).

axis : int or None, optional
: Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

Returns

mode : ndarray
: Array of modal values.

count : ndarray
: Array of counts for each mode.

Notes

For more details, see *stats.mode*.

`scipy.stats.mstats.`**`moment`**(*a*, *moment=1*, *axis=0*)

Calculates the nth moment about the mean for a sample.

Parameters

a : array_like
: data

moment : int, optional

order of central moment that is returned

axis : int or None, optional

Axis along which the central moment is computed. Default is 0. If None, compute over the whole array *a*.

Returns **n-th central moment** : ndarray or float

The appropriate moment along the given axis or over all values if axis is None. The denominator for the moment calculation is the number of observations, no degrees of freedom correction is done.

Notes

For more details about `moment`, see *stats.moment*.

`scipy.stats.mstats.`**`mquantiles`**(*a, prob=[0.25, 0.5, 0.75], alphap=0.4, betap=0.4, axis=None, limit=()*)

Computes empirical quantiles for a data array.

Samples quantile are defined by `Q(p) = (1-gamma)*x[j] + gamma*x[j+1]`, where `x[j]` is the j-th order statistic, and gamma is a function of `j = floor(n*p + m)`, `m = alphap + p*(1 - alphap - betap)` and `g = n*p + m - j`.

Reinterpreting the above equations to compare to **R** lead to the equation: `p(k) = (k - alphap)/(n + 1 - alphap - betap)`

Typical values of (alphap,betap) are:

- (0,1) : `p(k) = k/n` : linear interpolation of cdf (**R** type 4)
- (.5,.5) : `p(k) = (k - 1/2.)/n` : piecewise linear function (**R** type 5)
- (0,0) : `p(k) = k/(n+1)` : (**R** type 6)
- (1,1) : `p(k) = (k-1)/(n-1)`: p(k) = mode[F(x[k])]. (**R** type 7, **R** default)
- (1/3,1/3): `p(k) = (k-1/3)/(n+1/3)`: Then p(k) ~ median[F(x[k])]. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (**R** type 8)
- (3/8,3/8): `p(k) = (k-3/8)/(n+1/4)`: Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (**R** type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM

Parameters **a** : array_like

Input data, as a sequence or array of dimension at most 2.

prob : array_like, optional

List of quantiles to compute.

alphap : float, optional

Plotting positions parameter, default is 0.4.

betap : float, optional

Plotting positions parameter, default is 0.4.

axis : int, optional

Axis along which to perform the trimming. If None (default), the input array is first flattened.

limit : tuple, optional

Tuple of (lower, upper) values. Values of *a* outside this open interval are ignored.

Returns **mquantiles** : MaskedArray

An array containing the calculated quantiles.

Notes

This formulation is very similar to **R** except the calculation of `m` from `alphap` and `betap`, where in **R** `m` is defined with each type.

References

[R612], [R613]

Examples

```
>>> from scipy.stats.mstats import mquantiles
>>> a = np.array([6., 47., 49., 15., 42., 41., 7., 39., 43., 40., 36.])
>>> mquantiles(a)
array([ 19.2,  40. ,  42.8])
```

Using a 2D array, specifying axis and limit.

```
>>> data = np.array([[   6.,    7.,    1.],
...                  [  47.,   15.,    2.],
...                  [  49.,   36.,    3.],
...                  [  15.,   39.,    4.],
...                  [  42.,   40., -999.],
...                  [  41.,   41., -999.],
...                  [   7., -999., -999.],
...                  [  39., -999., -999.],
...                  [  43., -999., -999.],
...                  [  40., -999., -999.],
...                  [  36., -999., -999.]])
>>> print(mquantiles(data, axis=0, limit=(0, 50)))
[[ 19.2   14.6    1.45]
 [ 40.    37.5    2.5 ]
 [ 42.8   40.05   3.55]]
```

```
>>> data[:, 2] = -999.
>>> print(mquantiles(data, axis=0, limit=(0, 50)))
[[19.200000000000003 14.6 --]
 [40.0 37.5 --]
 [42.800000000000004 40.05 --]]
```

`scipy.stats.mstats.`**`msign`**(*x*)

Returns the sign of x, or 0 if x is masked.

`scipy.stats.mstats.`**`normaltest`**(*a, axis=0*)

Tests whether a sample differs from a normal distribution.

Parameters

a : array_like

The array containing the data to be tested.

axis : int or None, optional

Axis along which to compute test. Default is 0. If None, compute over the whole array *a*.

Returns

statistic : float or array

`s^2 + k^2`, where `s` is the z-score returned by *`skewtest`* and `k` is the z-score returned by *`kurtosistest`*.

pvalue : float or array

A 2-sided chi squared probability for the hypothesis test.

Notes

For more details about *`normaltest`*, see *stats.normaltest*.

`scipy.stats.mstats.`**`obrientransform`**(**args*)

Computes a transform on input data (any number of columns). Used to test for homogeneity of variance prior

to running one-way stats. Each array in `*args` is one level of a factor. If an *f_oneway()* run on the transformed data and found significant, variances are unequal. From Maxwell and Delaney, p.112.

Returns: transformed data for use in an ANOVA

`scipy.stats.mstats.`**`pearsonr`**(*x*, *y*)

Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

The Pearson correlation coefficient measures the linear relationship between two datasets. Strictly speaking, Pearson's correlation requires that each dataset be normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as *x* increases, so does *y*. Negative correlations imply that as *x* increases, *y* decreases.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters
x : 1-D array_like
    Input
y : 1-D array_like
    Input

Returns
pearsonr : float
    Pearson's correlation coefficient, 2-tailed p-value.

References

http://www.statsoft.com/textbook/glosp.html#Pearson%20Correlation

`scipy.stats.mstats.`**`pointbiserialr`**(*x*, *y*)

Calculates a point biserial correlation coefficient and its p-value.

Parameters
x : array_like of bools
    Input array.
y : array_like
    Input array.

Returns
correlation : float
    R value
pvalue : float
    2-tailed p-value

Notes

Missing values are considered pair-wise: if a value is missing in x, the corresponding value in y is masked.

For more details on *pointbiserialr*, see *stats.pointbiserialr*.

`scipy.stats.mstats.`**`rankdata`**(*data*, *axis=None*, *use_missing=False*)

Returns the rank (also known as order statistics) of each data point along the given axis.

If some values are tied, their rank is averaged. If some values are masked, their rank is set to 0 if use_missing is False, or set to the average rank of the unmasked values if use_missing is True.

Parameters
data : sequence
    Input data. The data is transformed to a masked array
axis : {None,int}, optional
    Axis along which to perform the ranking. If None, the array is first flattened. An exception is raised if the axis is specified for arrays with a dimension larger than 2
use_missing : bool, optional
    Whether the masked values have a rank of 0 (False) or equal to the average rank of the unmasked values (True).

`scipy.stats.mstats.`**`scoreatpercentile`**(*data*, *per*, *limit=()*, *alphap=0.4*, *betap=0.4*)

Calculate the score at the given 'per' percentile of the sequence a. For example, the score at per=50 is the median.

This function is a shortcut to mquantile

`scipy.stats.mstats.`**`sem`**(*a*, *axis=0*, *ddof=1*)

Calculates the standard error of the mean of the input array.

Also sometimes called standard error of measurement.

Parameters

a : array_like
    An array containing the values for which the standard error is returned.

axis : int or None, optional
    If axis is None, ravel *a* first. If axis is an integer, this will be the axis over which to operate. Defaults to 0.

ddof : int, optional
    Delta degrees-of-freedom. How many degrees of freedom to adjust for bias in limited samples relative to the population estimate of variance. Defaults to 1.

Returns

s : ndarray or float
    The standard error of the mean in the sample(s), along the input axis.

Notes

The default value for *ddof* changed in scipy 0.15.0 to be consistent with *stats.sem* as well as with the most common definition used (like in the R documentation).

Examples

Find standard error along the first axis:

```
>>> from scipy import stats
>>> a = np.arange(20).reshape(5,4)
>>> print(stats.mstats.sem(a))
[2.8284271247461903 2.8284271247461903 2.8284271247461903
 2.8284271247461903]
```

Find standard error across the whole array, using n degrees of freedom:

```
>>> print(stats.mstats.sem(a, axis=None, ddof=0))
1.2893796958227628
```

`scipy.stats.mstats.`**`signaltonoise`**(**args*, ***kwds*)

`signaltonoise` is deprecated! mstats.signaltonoise is deprecated in scipy 0.16.0

Calculates the signal-to-noise ratio, as the ratio of the mean over
    standard deviation along the given axis.

Parameters

data : sequence
    Input data

axis
    [{0, int}, optional] Axis along which to compute. If None, the computation is performed on a flat version of the array.

`scipy.stats.mstats.`**`skew`**(*a*, *axis=0*, *bias=True*)

Computes the skewness of a data set.

Parameters

a : ndarray
    data

axis : int or None, optional

Axis along which skewness is calculated. Default is 0. If None, compute over the whole array *a*.

bias : bool, optional
If False, then the calculations are corrected for statistical bias.

Returns **skewness** : ndarray
The skewness of values along an axis, returning 0 where all values are equal.

Notes

For more details about `skew`, see *stats.skew*.

`scipy.stats.mstats.`**`skewtest`**(*a*, *axis=0*)

Tests whether the skew is different from the normal distribution.

Parameters **a** : array
The data to be tested

axis : int or None, optional
Axis along which statistics are calculated. Default is 0. If None, compute over the whole array *a*.

Returns **statistic** : float
The computed z-score for this test.

pvalue : float
a 2-sided p-value for the hypothesis test

Notes

For more details about `skewtest`, see *stats.skewtest*.

`scipy.stats.mstats.`**`spearmanr`**(*x*, *y*, *use_ties=True*)

Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.

The Spearman correlation is a nonparametric measure of the linear relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as *x* increases, so does *y*. Negative correlations imply that as *x* increases, *y* decreases.

Missing values are discarded pair-wise: if a value is missing in *x*, the corresponding value in *y* is masked.

The p-value roughly indicates the probability of an uncorrelated system producing datasets that have a Spearman correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

Parameters **x** : array_like
The length of *x* must be > 2.

y : array_like
The length of *y* must be > 2.

use_ties : bool, optional
Whether the correction for ties should be computed.

Returns **correlation** : float
Spearman correlation coefficient

pvalue : float
2-tailed p-value.

References

[CRCProbStat2000] section 14.7

`scipy.stats.mstats.`**`theilslopes`**(*y*, *x=None*, *alpha=0.95*)

Computes the Theil-Sen estimator for a set of points (x, y).

`theilslopes` implements a method for robust linear regression. It computes the slope as the median of all slopes between paired values.

Parameters **y** : array_like
Dependent variable.
x : array_like or None, optional
Independent variable. If None, use `arange(len(y))` instead.
alpha : float, optional
Confidence degree between 0 and 1. Default is 95% confidence. Note that *alpha* is symmetric around 0.5, i.e. both 0.1 and 0.9 are interpreted as "find the 90% confidence interval".

Returns **medslope** : float
Theil slope.
medintercept : float
Intercept of the Theil line, as `median(y) - medslope*median(x)`.
lo_slope : float
Lower bound of the confidence interval on *medslope*.
up_slope : float
Upper bound of the confidence interval on *medslope*.

Notes

For more details on `theilslopes`, see *stats.theilslopes*.

`scipy.stats.mstats.`**`threshold`**`(*args, **kwds)`

`threshold` is deprecated! mstats.threshold is deprecated in scipy 0.17.0

Clip array to a given value.
Similar to numpy.clip(), except that values less than *threshmin* or greater than *threshmax* are replaced by *newval*, instead of by *threshmin* and *threshmax* respectively.

Parameters **a** : ndarray
Input data
threshmin [{None, float}, optional] Lower threshold. If None, set to the minimum value.
threshmax [{None, float}, optional] Upper threshold. If None, set to the maximum value.
newval [{0, float}, optional] Value outside the thresholds.

Returns **threshold** : ndarray
Returns *a*, with values less then *threshmin* and values greater *threshmax* replaced with *newval*.

`scipy.stats.mstats.`**`tmax`**`(a, upperlimit=None, axis=0, inclusive=True)`

Compute the trimmed maximum

This function computes the maximum value of an array along a given axis, while ignoring values larger than a specified upper limit.

Parameters **a** : array_like
array of values
upperlimit : None or float, optional
Values in the input array greater than the given limit will be ignored. When upperlimit is None, then all values are used. The default value is None.
axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.
inclusive : {True, False}, optional

This flag determines whether values exactly equal to the upper limit are included. The default value is True.

Returns **tmax** : float, int or ndarray

Notes

For more details on `tmax`, see *stats.tmax*.

`scipy.stats.mstats.`**`tmean`**(*a*, *limits=None*, *inclusive=(True*, *True)*, *axis=None*)

Compute the trimmed mean.

Parameters **a** : array_like

Array of values.

limits : None or (lower limit, upper limit), optional

Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None (default), then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval.

inclusive : (bool, bool), optional

A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional

Axis along which to operate. If None, compute over the whole array. Default is None.

Returns **tmean** : float

Notes

For more details on `tmean`, see *stats.tmean*.

`scipy.stats.mstats.`**`tmin`**(*a*, *lowerlimit=None*, *axis=0*, *inclusive=True*)

Compute the trimmed minimum

Parameters **a** : array_like

array of values

lowerlimit : None or float, optional

Values in the input array less than the given limit will be ignored. When lowerlimit is None, then all values are used. The default value is None.

axis : int or None, optional

Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

inclusive : {True, False}, optional

This flag determines whether values exactly equal to the lower limit are included. The default value is True.

Returns **tmin** : float, int or ndarray

Notes

For more details on `tmin`, see *stats.tmin*.

`scipy.stats.mstats.`**`trim`**(*a*, *limits=None*, *inclusive=(True*, *True)*, *relative=False*, *axis=None*)

Trims an array by masking the data outside some given limits.

Returns a masked version of the input array.

Parameters **a** : sequence

Input array

limits : {None, tuple}, optional

If *relative* is False, tuple (lower limit, upper limit) in absolute values. Values of the input array lower (greater) than the lower (upper) limit are masked.
If *relative* is True, tuple (lower percentage, upper percentage) to cut on each side of the array, with respect to the number of unmasked data.
Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) In each case, the value of one limit can be set to None to indicate an open interval.
If limits is None, no trimming is performed

inclusive : {(bool, bool) tuple}, optional
If *relative* is False, tuple indicating whether values exactly equal to the absolute limits are allowed. If *relative* is True, tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

relative : bool, optional
Whether to consider the limits as absolute values (False) or proportions to cut (True).

axis : int, optional
Axis along which to trim.

Examples

```
>>> from scipy.stats.mstats import trim
>>> z = [ 1, 2, 3, 4, 5, 6, 7, 8, 9,10]
>>> print(trim(z,(3,8)))
[-- -- 3 4 5 6 7 8 -- --]
>>> print(trim(z,(0.1,0.2),relative=True))
[-- 2 3 4 5 6 7 8 -- --]
```

`scipy.stats.mstats.`**`trima`**(*a*, *limits=None*, *inclusive=(True, True)*)
Trims an array by masking the data outside some given limits.

Returns a masked version of the input array.

Parameters

a : array_like
Input array.

limits : {None, tuple}, optional
Tuple of (lower limit, upper limit) in absolute values. Values of the input array lower (greater) than the lower (upper) limit will be masked. A limit is None indicates an open interval.

inclusive : (bool, bool) tuple, optional
Tuple of (lower flag, upper flag), indicating whether values exactly equal to the lower (upper) limit are allowed.

`scipy.stats.mstats.`**`trimboth`**(*data*, *proportiontocut=0.2*, *inclusive=(True, True)*, *axis=None*)
Trims the smallest and largest data values.

Trims the *data* by masking the `int(proportiontocut * n)` smallest and `int(proportiontocut * n)` largest values of data along the given axis, where n is the number of unmasked values before trimming.

Parameters

data : ndarray
Data to trim.

proportiontocut : float, optional
Percentage of trimming (as a float between 0 and 1). If n is the number of unmasked values before trimming, the number of values after trimming is `(1 - 2*proportiontocut) * n`. Default is 0.2.

inclusive : {(bool, bool) tuple}, optional

Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

axis : int, optional

Axis along which to perform the trimming. If None, the input array is first flattened.

`scipy.stats.mstats.`**`trimmed_stde`**`(`*a*, *limits=(0.1, 0.1)*, *inclusive=(1, 1)*, *axis=None*`)`

Returns the standard error of the trimmed mean along the given axis.

Parameters

a : sequence

Input array

limits : {(0.1,0.1), tuple of float}, optional

tuple (lower percentage, upper percentage) to cut on each side of the array, with respect to the number of unmasked data.

If n is the number of unmasked data before trimming, the values smaller than `n * limits[0]` and the values larger than `` n * `limits[1] `` are masked, and the total number of unmasked data after trimming is `n * (1. -sum(limits))`. In each case, the value of one limit can be set to None to indicate an open interval. If *limits* is None, no trimming is performed.

inclusive : {(bool, bool) tuple} optional

Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

axis : int, optional

Axis along which to trim.

Returns

trimmed_stde : scalar or ndarray

`scipy.stats.mstats.`**`trimr`**`(`*a*, *limits=None*, *inclusive=(True, True)*, *axis=None*`)`

Trims an array by masking some proportion of the data on each end. Returns a masked version of the input array.

Parameters

a : sequence

Input array.

limits : {None, tuple}, optional

Tuple of the percentages to cut on each side of the array, with respect to the number of unmasked data, as floats between 0. and 1. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)). The value of one limit can be set to None to indicate an open interval.

inclusive : {(True,True) tuple}, optional

Tuple of flags indicating whether the number of data being masked on the left (right) end should be truncated (True) or rounded (False) to integers.

axis : {None,int}, optional

Axis along which to trim. If None, the whole array is trimmed, but its shape is maintained.

`scipy.stats.mstats.`**`trimtail`**`(`*data*, *proportiontocut=0.2*, *tail='left'*, *inclusive=(True, True)*, *axis=None*`)`

Trims the data by masking values from one tail.

Parameters

data : array_like

Data to trim.

proportiontocut : float, optional

Percentage of trimming. If `n` is the number of unmasked values before trimming, the number of values after trimming is `(1 - proportiontocut) * n`. Default is 0.2.

tail : {'left','right'}, optional

If 'left' the *proportiontocut* lowest values will be masked. If 'right' the *proportiontocut* highest values will be masked. Default is 'left'.

inclusive : {(bool, bool) tuple}, optional
Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False). Default is (True, True).

axis : int, optional
Axis along which to perform the trimming. If None, the input array is first flattened. Default is None.

Returns **trimtail** : ndarray
Returned array of same shape as *data* with masked tail values.

`scipy.stats.mstats.`**`tsem`**(*a*, *limits=None*, *inclusive=(True*, *True)*, *axis=0*, *ddof=1*)

Compute the trimmed standard error of the mean.

This function finds the standard error of the mean for given values, ignoring values outside the given *limits*.

Parameters **a** : array_like
array of values

limits : None or (lower limit, upper limit), optional
Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval. The default value is None.

inclusive : (bool, bool), optional
A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
Axis along which to operate. If None, compute over the whole array. Default is zero.

ddof : int, optional
Delta degrees of freedom. Default is 1.

Returns **tsem** : float

Notes

For more details on `tsem`, see *stats.tsem*.

`scipy.stats.mstats.`**`ttest_onesamp`**(*a*, *popmean*, *axis=0*)

Calculates the T-test for the mean of ONE group of scores.

Parameters **a** : array_like
sample observation

popmean : float or array_like
expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int or None, optional
Axis along which to compute test. If None, compute over the whole array *a*.

Returns **statistic** : float or array
t-statistic

pvalue : float or array
two-tailed p-value

Notes

For more details on `ttest_1samp`, see *stats.ttest_1samp*.

`scipy.stats.mstats.`**`ttest_ind`**(*a*, *b*, *axis=0*, *equal_var=True*)

Calculates the T-test for the means of TWO INDEPENDENT samples of scores.

Parameters

a, b : array_like
: The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int or None, optional
: Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.

equal_var : bool, optional
: If True, perform a standard independent 2 sample test that assumes equal population variances. If False, perform Welch's t-test, which does not assume equal population variance. .. versionadded:: 0.17.0

Returns

statistic : float or array
: The calculated t-statistic.

pvalue : float or array
: The two-tailed p-value.

Notes

For more details on `ttest_ind`, see *stats.ttest_ind*.

`scipy.stats.mstats.`**`ttest_rel`**(*a*, *b*, *axis=0*)

Calculates the T-test on TWO RELATED samples of scores, a and b.

Parameters

a, b : array_like
: The arrays must have the same shape.

axis : int or None, optional
: Axis along which to compute test. If None, compute over the whole arrays, *a*, and *b*.

Returns

statistic : float or array
: t-statistic

pvalue : float or array
: two-tailed p-value

Notes

For more details on `ttest_rel`, see *stats.ttest_rel*.

`scipy.stats.mstats.`**`tvar`**(*a*, *limits=None*, *inclusive=(True, True)*, *axis=0*, *ddof=1*)

Compute the trimmed variance

This function computes the sample variance of an array of values, while ignoring values which are outside of given *limits*.

Parameters

a : array_like
: Array of values.

limits : None or (lower limit, upper limit), optional
: Values in the input array less than the lower limit or greater than the upper limit will be ignored. When limits is None, then all values are used. Either of the limit values in the tuple can also be None representing a half-open interval. The default value is None.

inclusive : (bool, bool), optional
: A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).

axis : int or None, optional
: Axis along which to operate. If None, compute over the whole array. Default is zero.

ddof : int, optional
: Delta degrees of freedom. Default is 1.

Returns

tvar : float

Trimmed variance.

Notes

For more details on `tvar`, see *stats.tvar*.

`scipy.stats.mstats.`**`variation`**(*a*, *axis=0*)

Computes the coefficient of variation, the ratio of the biased standard deviation to the mean.

Parameters

a : array_like
    Input array.

axis : int or None, optional
    Axis along which to calculate the coefficient of variation. Default is 0. If None, compute over the whole array *a*.

Returns

variation : ndarray
    The calculated variation along the requested axis.

Notes

For more details about `variation`, see *stats.variation*.

`scipy.stats.mstats.`**`winsorize`**(*a*, *limits=None*, *inclusive=(True*, *True)*, *inplace=False*, *axis=None*)

Returns a Winsorized version of the input array.

The (limits[0])th lowest values are set to the (limits[0])th percentile, and the (limits[1])th highest values are set to the (1 - limits[1])th percentile. Masked values are skipped.

Parameters

a : sequence
    Input array.

limits : {None, tuple of float}, optional
    Tuple of the percentages to cut on each side of the array, with respect to the number of unmasked data, as floats between 0. and 1. Noting n the number of unmasked data before trimming, the (n*limits[0])th smallest data and the (n*limits[1])th largest data are masked, and the total number of unmasked data after trimming is n*(1.-sum(limits)) The value of one limit can be set to None to indicate an open interval.

inclusive : {(True, True) tuple}, optional
    Tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).

inplace : {False, True}, optional
    Whether to winsorize in place (True) or to use a copy (False)

axis : {None, int}, optional
    Axis along which to trim. If None, the whole array is trimmed, but its shape is maintained.

Notes

This function is applied to reduce the effect of possibly spurious outliers by limiting the extreme values.

`scipy.stats.mstats.`**`zmap`**(*scores*, *compare*, *axis=0*, *ddof=0*)

Calculates the relative z-scores.

Returns an array of z-scores, i.e., scores that are standardized to zero mean and unit variance, where mean and variance are calculated from the comparison array.

Parameters

scores : array_like
    The input for which z-scores are calculated.

compare : array_like
    The input from which the mean and standard deviation of the normalization are taken; assumed to have the same dimension as *scores*.

axis : int or None, optional
Axis over which mean and variance of *compare* are calculated. Default is 0. If None, compute over the whole array *scores*.

ddof : int, optional
Degrees of freedom correction in the calculation of the standard deviation. Default is 0.

Returns **zscore** : array_like
Z-scores, in the same shape as *scores*.

Notes

This function preserves ndarray subclasses, and works also with matrices and masked arrays (it uses *asanyarray* instead of *asarray* for parameters).

Examples

```
>>> from scipy.stats import zmap
>>> a = [0.5, 2.0, 2.5, 3]
>>> b = [0, 1, 2, 3, 4]
>>> zmap(a, b)
array([-1.06066017,  0.        ,  0.35355339,  0.70710678])
```

`scipy.stats.mstats.`**`zscore`**`(`*a*, *axis=0*, *ddof=0*`)`
Calculates the z score of each value in the sample, relative to the sample mean and standard deviation.

Parameters **a** : array_like
An array like object containing the sample data.

axis : int or None, optional
Axis along which to operate. Default is 0. If None, compute over the whole array *a*.

ddof : int, optional
Degrees of freedom correction in the calculation of the standard deviation. Default is 0.

Returns **zscore** : array_like
The z-scores, standardized by mean and standard deviation of input array *a*.

Notes

This function preserves ndarray subclasses, and works also with matrices and masked arrays (it uses *asanyarray* instead of *asarray* for parameters).

Examples

```
>>> a = np.array([ 0.7972,  0.0767,  0.4383,  0.7866,  0.8091,
...                0.1954,  0.6307,  0.6599,  0.1065,  0.0508])
>>> from scipy import stats
>>> stats.zscore(a)
array([ 1.1273, -1.247 , -0.0552,  1.0923,  1.1664, -0.8559,  0.5786,
        0.6748, -1.1488, -1.3324])
```

Computing along a specified axis, using n-1 degrees of freedom (`ddof=1`) to calculate the standard deviation:

```
>>> b = np.array([[ 0.3148,  0.0478,  0.6243,  0.4608],
...               [ 0.7149,  0.0775,  0.6072,  0.9656],
...               [ 0.6341,  0.1403,  0.9759,  0.4064],
...               [ 0.5918,  0.6948,  0.904 ,  0.3721],
...               [ 0.0921,  0.2481,  0.1188,  0.1366]])
>>> stats.zscore(b, axis=1, ddof=1)
array([[-0.19264823, -1.28415119,  1.07259584,  0.40420358],
```

```
[ 0.33048416, -1.37380874,  0.04251374,  1.00081084],
[ 0.26796377, -1.12598418,  1.23283094, -0.37481053],
[-0.22095197,  0.24468594,  1.19042819, -1.21416216],
[-0.82780366,  1.4457416 , -0.43867764, -0.1792603 ]])
```

scipy.stats.mstats.**compare_medians_ms**(*group_1*, *group_2*, *axis=None*)

Compares the medians from two independent groups along the given axis.

The comparison is performed using the McKean-Schrader estimate of the standard error of the medians.

Parameters **group_1** : array_like
First dataset.

group_2 : array_like
Second dataset.

axis : int, optional
Axis along which the medians are estimated. If None, the arrays are flattened. If *axis* is not None, then *group_1* and *group_2* should have the same shape.

Returns **compare_medians_ms** : {float, ndarray}
If *axis* is None, then returns a float, otherwise returns a 1-D ndarray of floats with a length equal to the length of *group_1* along *axis*.

scipy.stats.mstats.**gmean**(*a*, *axis=0*, *dtype=None*)

Compute the geometric mean along the specified axis.

Returns the geometric average of the array elements. That is: n-th root of (x1 * x2 * ... * xn)

Parameters **a** : array_like
Input array or object that can be converted to an array.

axis : int or None, optional
Axis along which the geometric mean is computed. Default is 0. If None, compute over the whole array *a*.

dtype : dtype, optional
Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

Returns **gmean** : ndarray
see dtype parameter above

See also:

`numpy.mean` Arithmetic average

`numpy.average` Weighted average

hmean Harmonic mean

Notes

The geometric average is computed over a single dimension of the input array, axis=0 by default, or all values in the array if axis=None. float64 intermediate and return values are used for integer inputs.

Use masked arrays to ignore any non-finite values in the input or that arise in the calculations such as Not a Number and infinity because masked arrays automatically mask any non-finite values.

scipy.stats.mstats.**hdmedian**(*data*, *axis=-1*, *var=False*)

Returns the Harrell-Davis estimate of the median along the given axis.

Parameters **data** : ndarray
Data array.

axis : int, optional
Axis along which to compute the quantiles. If None, use a flattened array.
var : bool, optional
Whether to return the variance of the estimate.

`scipy.stats.mstats.hdquantiles`(*data, prob=[0.25, 0.5, 0.75], axis=None, var=False*)
Computes quantile estimates with the Harrell-Davis method.

The quantile estimates are calculated as a weighted linear combination of order statistics.

Parameters
data : array_like
Data array.
prob : sequence, optional
Sequence of quantiles to compute.
axis : int or None, optional
Axis along which to compute the quantiles. If None, use a flattened array.
var : bool, optional
Whether to return the variance of the estimate.

Returns
hdquantiles : MaskedArray
A (p,) array of quantiles (if *var* is False), or a (2,p) array of quantiles and variances (if *var* is True), where `p` is the number of quantiles.

`scipy.stats.mstats.hdquantiles_sd`(*data, prob=[0.25, 0.5, 0.75], axis=None*)
The standard error of the Harrell-Davis quantile estimates by jackknife.

Parameters
data : array_like
Data array.
prob : sequence, optional
Sequence of quantiles to compute.
axis : int, optional
Axis along which to compute the quantiles. If None, use a flattened array.

Returns
hdquantiles_sd : MaskedArray
Standard error of the Harrell-Davis quantile estimates.

`scipy.stats.mstats.hmean`(*a, axis=0, dtype=None*)
Calculates the harmonic mean along the specified axis.

That is: n / (1/x1 + 1/x2 + ... + 1/xn)

Parameters
a : array_like
Input array, masked array or object that can be converted to an array.
axis : int or None, optional
Axis along which the harmonic mean is computed. Default is 0. If None, compute over the whole array *a*.
dtype : dtype, optional
Type of the returned array and of the accumulator in which the elements are summed. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer *dtype* with a precision less than that of the default platform integer. In that case, the default platform integer is used.

Returns
hmean : ndarray
see *dtype* parameter above

See also:

`numpy.mean` Arithmetic average
`numpy.average` Weighted average
`gmean` Geometric mean

Notes

The harmonic mean is computed over a single dimension of the input array, axis=0 by default, or all values in the array if axis=None. float64 intermediate and return values are used for integer inputs.

Use masked arrays to ignore any non-finite values in the input or that arise in the calculations such as Not a Number and infinity.

`scipy.stats.mstats.idealfourths(data, axis=None)`

Returns an estimate of the lower and upper quartiles.

Uses the ideal fourths algorithm.

Parameters
- **data** : array_like
  Input array.
- **axis** : int, optional
  Axis along which the quartiles are estimated. If None, the arrays are flattened.

Returns
- **idealfourths** : {list of floats, masked array}
  Returns the two internal values that divide *data* into four parts using the ideal fourths algorithm either along the flattened array (if *axis* is None) or along *axis* of *data*.

`scipy.stats.mstats.kruskal(*args)`

Compute the Kruskal-Wallis H-test for independent samples

Parameters
- **sample1, sample2, ...** : array_like
  Two or more arrays with the sample measurements can be given as arguments.

Returns
- **statistic** : float
  The Kruskal-Wallis H statistic, corrected for ties
- **pvalue** : float
  The p-value for the test using the assumption that H has a chi square distribution

Notes

For more details on `kruskal`, see *stats.kruskal*.

`scipy.stats.mstats.ks_2samp(data1, data2, alternative='two-sided')`

Computes the Kolmogorov-Smirnov test on two samples.

Missing values are discarded.

Parameters
- **data1** : array_like
  First data set
- **data2** : array_like
  Second data set
- **alternative** : {'two-sided', 'less', 'greater'}, optional
  Indicates the alternative hypothesis. Default is 'two-sided'.

Returns
- **d** : float
  Value of the Kolmogorov Smirnov test
- **p** : float
  Corresponding p-value.

`scipy.stats.mstats.median_cihs(data, alpha=0.05, axis=None)`

Computes the alpha-level confidence interval for the median of the data.

Uses the Hettmasperger-Sheather method.

Parameters
- **data** : array_like

Input data. Masked values are discarded. The input should be 1D only, or *axis* should be set to None.

alpha : float, optional

Confidence level of the intervals.

axis : int or None, optional

Axis along which to compute the quantiles. If None, use a flattened array.

Returns median_cihs

Alpha level confidence interval.

`scipy.stats.mstats.`**`meppf`**`(`*data*, *alpha=0.4*, *beta=0.4*`)`

Returns plotting positions (or empirical percentile points) for the data.

Plotting positions are defined as **`(i-alpha)/(n+1-alpha-beta)`*****, where:***

- i is the rank order statistics
- n is the number of unmasked values along the given axis
- *alpha* and *beta* are two parameters.

Typical values for **alpha** ***and*** **beta** ***are:***

- (0,1) : `p(k) = k/n`, linear interpolation of cdf (R, type 4)
- (.5,.5) : `p(k) = (k-1/2.)/n`, piecewise linear function (R, type 5)
- (0,0) : `p(k) = k/(n+1)`, Weibull (R type 6)
- (1,1) : `p(k) = (k-1)/(n-1)`, in this case, `p(k) = mode[F(x[k])]`. That's R default (R type 7)
- (1/3,1/3): `p(k) = (k-1/3)/(n+1/3)`, then `p(k) ~ median[F(x[k])]`. The resulting quantile estimates are approximately median-unbiased regardless of the distribution of x. (R type 8)
- (3/8,3/8): `p(k) = (k-3/8)/(n+1/4)`, Blom. The resulting quantile estimates are approximately unbiased if x is normally distributed (R type 9)
- (.4,.4) : approximately quantile unbiased (Cunnane)
- (.35,.35): APL, used with PWM
- (.3175, .3175): used in scipy.stats.probplot

Parameters **data** : array_like

Input data, as a sequence or array of dimension at most 2.

alpha : float, optional

Plotting positions parameter. Default is 0.4.

beta : float, optional

Plotting positions parameter. Default is 0.4.

Returns **positions** : MaskedArray

The calculated plotting positions.

`scipy.stats.mstats.`**`mjci`**`(`*data, prob=[0.25, 0.5, 0.75], axis=None*`)`

Returns the Maritz-Jarrett estimators of the standard error of selected experimental quantiles of the data.

Parameters **data** : ndarray

Data array.

prob : sequence, optional

Sequence of quantiles to compute.

axis : int or None, optional

Axis along which to compute the quantiles. If None, use a flattened array.

`scipy.stats.mstats.`**`mquantiles_cimj`**`(`*data, prob=[0.25, 0.5, 0.75], alpha=0.05, axis=None*`)`

Computes the alpha confidence interval for the selected quantiles of the data, with Maritz-Jarrett estimators.

Parameters **data** : ndarray

Data array.

prob : sequence, optional

Sequence of quantiles to compute.

alpha : float, optional
Confidence level of the intervals.

axis : int or None, optional
Axis along which to compute the quantiles. If None, use a flattened array.

`scipy.stats.mstats.`**`rsh`**(*data*, *points=None*)

Evaluates Rosenblatt's shifted histogram estimators for each point on the dataset 'data'.

Parameters **data** : sequence
Input data. Masked values are ignored.

points : sequence or None, optional
Sequence of points where to evaluate Rosenblatt shifted histogram. If None, use the data.

`scipy.stats.mstats.`**`sen_seasonal_slopes`**(*x*)

`scipy.stats.mstats.`**`trimmed_mean`**(*a*, *limits=(0.1, 0.1)*, *inclusive=(1, 1)*, *relative=True*, *axis=None*)

`scipy.stats.mstats.`**`trimmed_mean_ci`**(*data*, *limits=(0.2, 0.2)*, *inclusive=(True, True)*, *alpha=0.05*, *axis=None*)

Selected confidence interval of the trimmed mean along the given axis.

Parameters **data** : array_like
Input data.

limits : {None, tuple}, optional
None or a two item tuple. Tuple of the percentages to cut on each side of the array, with respect to the number of unmasked data, as floats between 0. and 1. If `n` is the number of unmasked data before trimming, then (`n * limits[0]`)th smallest data and (`n * limits[1]`)th largest data are masked. The total number of unmasked data after trimming is `n * (1. - sum(limits))`. The value of one limit can be set to None to indicate an open interval.
Defaults to (0.2, 0.2).

inclusive : (2,) tuple of boolean, optional
If relative==False, tuple indicating whether values exactly equal to the absolute limits are allowed. If relative==True, tuple indicating whether the number of data being masked on each side should be rounded (True) or truncated (False).
Defaults to (True, True).

alpha : float, optional
Confidence level of the intervals.
Defaults to 0.05.

axis : int, optional
Axis along which to cut. If None, uses a flattened version of *data*.
Defaults to None.

Returns **trimmed_mean_ci** : (2,) ndarray
The lower and upper confidence intervals of the trimmed data.

`scipy.stats.mstats.`**`trimmed_std`**(*a*, *limits=(0.1, 0.1)*, *inclusive=(1, 1)*, *relative=True*, *axis=None*, *ddof=0*)

`scipy.stats.mstats.`**`trimmed_var`**(*a*, *limits=(0.1, 0.1)*, *inclusive=(1, 1)*, *relative=True*, *axis=None*, *ddof=0*)

scipy.stats.mstats.**ttest_1samp**(*a*, *popmean*, *axis=0*)

Calculates the T-test for the mean of ONE group of scores.

Parameters

a : array_like
    sample observation

popmean : float or array_like
    expected value in null hypothesis, if array_like than it must have the same shape as *a* excluding the axis dimension

axis : int or None, optional
    Axis along which to compute test. If None, compute over the whole array *a*.

Returns

statistic : float or array
    t-statistic

pvalue : float or array
    two-tailed p-value

Notes

For more details on `ttest_1samp`, see *stats.ttest_1samp*.

5.27.9 Univariate and multivariate kernel density estimation (`scipy.stats.kde`)

gaussian_kde(dataset[, bw_method])	Representation of a kernel-density estimate using Gaussian kernels.

class scipy.stats.**gaussian_kde**(*dataset*, *bw_method=None*)

Representation of a kernel-density estimate using Gaussian kernels.

Kernel density estimation is a way to estimate the probability density function (PDF) of a random variable in a non-parametric way. `gaussian_kde` works for both uni-variate and multi-variate data. It includes automatic bandwidth determination. The estimation works best for a unimodal distribution; bimodal or multi-modal distributions tend to be oversmoothed.

Parameters

dataset : array_like
    Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2-D array with shape (# of dims, # of data).

bw_method : str, scalar or callable, optional
    The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as *kde.factor*. If a callable, it should take a `gaussian_kde` instance as only parameter and return a scalar. If None (default), 'scott' is used. See Notes for more details.

Notes

Bandwidth selection strongly influences the estimate obtained from the KDE (much more so than the actual shape of the kernel). Bandwidth selection can be done by a "rule of thumb", by cross-validation, by "plug-in methods" or by other means; see *[R585]*, *[R586]* for reviews. `gaussian_kde` uses a rule of thumb, the default is Scott's Rule.

Scott's Rule *[R583]*, implemented as `scotts_factor`, is:

```
n**(-1./(d+4)),
```

with `n` the number of data points and `d` the number of dimensions. Silverman's Rule *[R584]*, implemented as `silverman_factor`, is:

```
(n * (d + 2) / 4.)**(-1. / (d + 4)).
```

Good general descriptions of kernel density estimation can be found in *[R583]* and *[R584]*, the mathematics for this multi-dimensional implementation can be found in *[R583]*.

References

[R583], *[R584]*, *[R585]*, *[R586]*

Examples

Generate some random two-dimensional data:

```
>>> from scipy import stats
>>> def measure(n):
...     "Measurement model, return two coupled measurements."
...     m1 = np.random.normal(size=n)
...     m2 = np.random.normal(scale=0.5, size=n)
...     return m1+m2, m1-m2
```

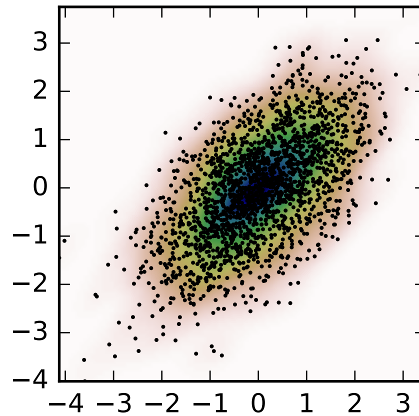
```
>>> m1, m2 = measure(2000)
>>> xmin = m1.min()
>>> xmax = m1.max()
>>> ymin = m2.min()
>>> ymax = m2.max()
```

Perform a kernel density estimate on the data:

```
>>> X, Y = np.mgrid[xmin:xmax:100j, ymin:ymax:100j]
>>> positions = np.vstack([X.ravel(), Y.ravel()])
>>> values = np.vstack([m1, m2])
>>> kernel = stats.gaussian_kde(values)
>>> Z = np.reshape(kernel(positions).T, X.shape)
```

Plot the results:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.rot90(Z), cmap=plt.cm.gist_earth_r,
...           extent=[xmin, xmax, ymin, ymax])
>>> ax.plot(m1, m2, 'k.', markersize=2)
>>> ax.set_xlim([xmin, xmax])
>>> ax.set_ylim([ymin, ymax])
>>> plt.show()
```



Attributes

dataset	(ndarray) The dataset with which `gaussian_kde` was initialized.
d	(int) Number of dimensions.
n	(int) Number of datapoints.
factor	(float) The bandwidth factor, obtained from *kde.covariance_factor*, with which the covariance matrix is multiplied.
covariance	(ndarray) The covariance matrix of *dataset*, scaled by the calculated bandwidth (*kde.factor*).
inv_cov	(ndarray) The inverse of *covariance*.

Methods

`evaluate`(points)	Evaluate the estimated pdf on a set of points.
`__call__`(points)	Evaluate the estimated pdf on a set of points.
`integrate_gaussian`(mean, cov)	Multiply estimated density by a multivariate Gaussian and integrate over the whole space.
`integrate_box_1d`(low, high)	Computes the integral of a 1D pdf between two bounds.
`integrate_box`(low_bounds, high_bounds[, maxpts])	Computes the integral of a pdf over a rectangular interval.
`integrate_kde`(other)	Computes the integral of the product of this kernel density estimate with another.
`pdf`(x)	Evaluate the estimated pdf on a provided set of points.
`logpdf`(x)	Evaluate the log of the estimated pdf on a provided set of points.
`resample`([size])	Randomly sample a dataset from the estimated pdf.
`set_bandwidth`([bw_method])	Compute the estimator bandwidth with given method.
`covariance_factor`()	Computes the coefficient (*kde.factor*) that multiplies the data covariance matrix to obtain the kernel covariance matrix.

`gaussian_kde.`**`evaluate`**(*points*)

Evaluate the estimated pdf on a set of points.

Parameters **points** : (# of dimensions, # of points)-array
Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.

Returns **values** : (# of points,)-array
The values at each point.

Raises **ValueError** : if the dimensionality of the input points is different than the dimensionality of the KDE.

`gaussian_kde.__call__`(*points*)

Evaluate the estimated pdf on a set of points.

Parameters **points** : (# of dimensions, # of points)-array
Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.

Returns **values** : (# of points,)-array
The values at each point.

Raises **ValueError** : if the dimensionality of the input points is different than the dimensionality of the KDE.

`gaussian_kde.integrate_gaussian`(*mean*, *cov*)

Multiply estimated density by a multivariate Gaussian and integrate over the whole space.

Parameters **mean** : aray_like
A 1-D array, specifying the mean of the Gaussian.

cov : array_like
A 2-D array, specifying the covariance matrix of the Gaussian.

Returns **result** : scalar
The value of the integral.

Raises **ValueError**
If the mean or covariance of the input Gaussian differs from the KDE's dimensionality.

`gaussian_kde.integrate_box_1d`(*low*, *high*)

Computes the integral of a 1D pdf between two bounds.

Parameters **low** : scalar
Lower bound of integration.

high : scalar
Upper bound of integration.

Returns **value** : scalar
The result of the integral.

Raises **ValueError**
If the KDE is over more than one dimension.

`gaussian_kde.integrate_box`(*low_bounds*, *high_bounds*, *maxpts=None*)

Computes the integral of a pdf over a rectangular interval.

Parameters **low_bounds** : array_like
A 1-D array containing the lower bounds of integration.

high_bounds : array_like
A 1-D array containing the upper bounds of integration.

maxpts : int, optional
The maximum number of points to use for integration.

Returns **value** : scalar
The result of the integral.

`gaussian_kde.integrate_kde`(*other*)

Computes the integral of the product of this kernel density estimate with another.

Parameters **other** : gaussian_kde instance
The other kde.

Returns **value** : scalar
The result of the integral.

Raises **ValueError**

If the KDEs have different dimensionality.

`gaussian_kde.`**`pdf`**`(`*x*`)`

Evaluate the estimated pdf on a provided set of points.

Notes

This is an alias for `gaussian_kde.evaluate`. See the `evaluate` docstring for more details.

`gaussian_kde.`**`logpdf`**`(`*x*`)`

Evaluate the log of the estimated pdf on a provided set of points.

`gaussian_kde.`**`resample`**`(`*size=None*`)`

Randomly sample a dataset from the estimated pdf.

Parameters **size** : int, optional

The number of samples to draw. If not provided, then the size is the same as the underlying dataset.

Returns **resample** : (self.d, *size*) ndarray

The sampled dataset.

`gaussian_kde.`**`set_bandwidth`**`(`*bw_method=None*`)`

Compute the estimator bandwidth with given method.

The new bandwidth calculated after a call to `set_bandwidth` is used for subsequent evaluations of the estimated density.

Parameters **bw_method** : str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as *kde.factor*. If a callable, it should take a `gaussian_kde` instance as only parameter and return a scalar. If None (default), nothing happens; the current *kde.covariance_factor* method is kept.

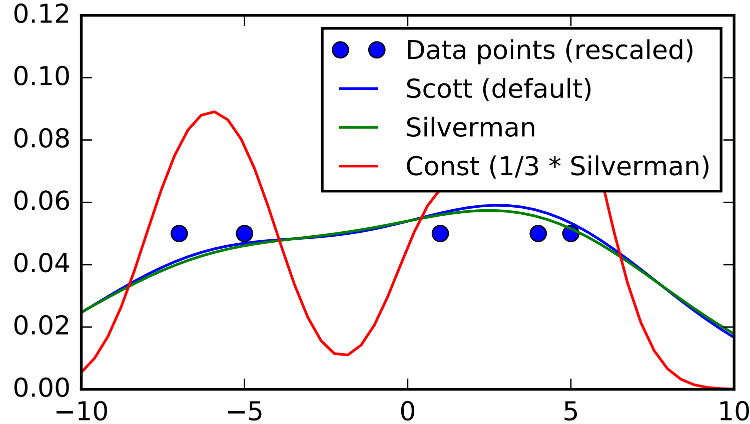
Notes

New in version 0.11.

Examples

```
>>> import scipy.stats as stats
>>> x1 = np.array([-7, -5, 1, 4, 5.])
>>> kde = stats.gaussian_kde(x1)
>>> xs = np.linspace(-10, 10, num=50)
>>> y1 = kde(xs)
>>> kde.set_bandwidth(bw_method='silverman')
>>> y2 = kde(xs)
>>> kde.set_bandwidth(bw_method=kde.factor / 3.)
>>> y3 = kde(xs)
```

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots()
>>> ax.plot(x1, np.ones(x1.shape) / (4. * x1.size), 'bo',
...         label='Data points (rescaled)')
>>> ax.plot(xs, y1, label='Scott (default)')
>>> ax.plot(xs, y2, label='Silverman')
>>> ax.plot(xs, y3, label='Const (1/3 * Silverman)')
>>> ax.legend()
>>> plt.show()
```



gaussian_kde.**covariance_factor**()

Computes the coefficient (*kde.factor*) that multiplies the data covariance matrix to obtain the kernel covariance matrix. The default is `scotts_factor`. A subclass can overwrite this method to provide a different method, or set it through a call to *kde.set_bandwidth*.

For many more stat related functions install the software R and the interface package rpy.

5.28 Low-level callback functions

Some functions in SciPy take as arguments callback functions, which can either be python callables or low-level compiled functions. Using compiled callback functions can improve performance somewhat by avoiding wrapping data in Python objects.

Such low-level functions in Scipy are wrapped in *LowLevelCallable* objects, which can be constructed from function pointers obtained from ctypes, cffi, Cython, or contained in Python *PyCapsule* objects.

LowLevelCallable	Low-level callback function.

class scipy.**LowLevelCallable**

Low-level callback function.

Parameters **function** : {PyCapsule, ctypes function pointer, cffi function pointer}
Low-level callback function.
user_data : {PyCapsule, ctypes void pointer, cffi void pointer}
User data to pass on to the callback function.
signature : str, optional
Signature of the function. If omitted, determined from *function*, if possible.

Notes

The argument `function` can be one of:

- PyCapsule, whose name contains the C function signature
- ctypes function pointer
- cffi function pointer

The signature of the low-level callback must match one of those expected by the routine it is passed to.

If constructing low-level functions from a PyCapsule, the name of the capsule must be the corresponding signature, in the format:

```
return_type (arg1_type, arg2_type, ...)
```

For example:

```
"void (double)"
"double (double, int *, void *)"
```

The context of a PyCapsule passed in as `function` is used as `user_data`, if an explicit value for `user_data` was not given.

Attributes

function	Callback function given
user_data	User data given
signature	Signature of the function.

Methods

from_cython(module, name[, user_data, signature])	Create a low-level callback function from an exported Cython function.

classmethod `LowLevelCallable.`**`from_cython`**(*module*, *name*, *user_data=None*, *signature=None*)

Create a low-level callback function from an exported Cython function.

Parameters **module** : module

Cython module where the exported function resides

name : str

Name of the exported function

user_data : {PyCapsule, ctypes void pointer, cffi void pointer}, optional

User data to pass on to the callback function.

signature : str, optional

Signature of the function. If omitted, determined from *function*.

See also:

Functions accepting low-level callables:

scipy.integrate.quad, *scipy.ndimage.generic_filter*, *scipy.ndimage.generic_filter1d*, *scipy.ndimage.geometric_transform*

Usage examples:

Extending scipy.ndimage in C, *Faster integration using low-level callback functions*

BIBLIOGRAPHY

[WPR] https://en.wikipedia.org/wiki/Romberg's_method

[NPT] https://docs.scipy.org/doc/numpy/reference/generated/numpy.trapz.html

[MOL] https://en.wikipedia.org/wiki/Method_of_lines

[KK] D.A. Knoll and D.E. Keyes, "Jacobian-free Newton-Krylov methods", J. Comp. Phys. 193, 357 (2004). doi:10.1016/j.jcp.2003.08.010

[PP] PETSc http://www.mcs.anl.gov/petsc/ and its Python bindings http://code.google.com/p/petsc4py/

[AMG] PyAMG (algebraic multigrid preconditioners/solvers) http://code.google.com/p/pyamg/

[CT65] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19: 297-301.

[NR07] Press, W., Teukolsky, S., Vetterline, W.T., and Flannery, B.P., 2007, *Numerical Recipes: The Art of Scientific Computing*, ch. 12-13. Cambridge Univ. Press, Cambridge, UK.

[Mak] J. Makhoul, 1980, 'A Fast Cosine Transform in One and Two Dimensions', *IEEE Transactions on acoustics, speech and signal processing* vol. 28(1), pp. 27-34, http://dx.doi.org/10.1109/TASSP.1980.1163351

[WPW] http://en.wikipedia.org/wiki/Window_function

[WPC] http://en.wikipedia.org/wiki/Discrete_cosine_transform

[WPS] http://en.wikipedia.org/wiki/Discrete_sine_transform

[R41] Daniel Mullner, "Modern hierarchical, agglomerative clustering algorithms", arXiv:1109.2378v1.

[R1] "Statistics toolbox." API Reference Documentation. The MathWorks. http://www.mathworks.com/access/helpdesk/help/toolbox/stats/. Accessed October 1, 2007.

[R2] "Hierarchical clustering." API Reference Documentation. The Wolfram Research, Inc. http://reference.wolfram.com/mathematica/HierarchicalClustering/tutorial/ HierarchicalClustering.html. Accessed October 1, 2007.

[R3] Gower, JC and Ross, GJS. "Minimum Spanning Trees and Single Linkage Cluster Analysis." Applied Statistics. 18(1): pp. 54–64. 1969.

[R4] Ward Jr, JH. "Hierarchical grouping to optimize an objective function." Journal of the American Statistical Association. 58(301): pp. 236–44. 1963.

[R5] Johnson, SC. "Hierarchical clustering schemes." Psychometrika. 32(2): pp. 241–54. 1966.

[R6] Sneath, PH and Sokal, RR. "Numerical taxonomy." Nature. 193: pp. 855–60. 1962.

[R7] Batagelj, V. "Comparing resemblance measures." Journal of Classification. 12: pp. 73–90. 1995.

[R8] Sokal, RR and Michener, CD. "A statistical method for evaluating systematic relationships." Scientific Bulletins. 38(22): pp. 1409–38. 1958.

[R9] Edelbrock, C. "Mixture model tests of hierarchical clustering algorithms: the problem of classifying everybody." Multivariate Behavioral Research. 14: pp. 367–84. 1979.

[CODATA2014] CODATA Recommended Values of the Fundamental Physical Constants 2014.

http://physics.nist.gov/cuu/Constants/index.html

[R42] 'A Fast Cosine Transform in One and Two Dimensions', by J. Makhoul, *IEEE Transactions on acoustics, speech and signal processing* vol. 28(1), pp. 27-34, http://dx.doi.org/10.1109/TASSP.1980.1163351 (1980).

[R43] Wikipedia, "Discrete cosine transform", http://en.wikipedia.org/wiki/Discrete_cosine_transform

[R44] Wikipedia, "Discrete sine transform", http://en.wikipedia.org/wiki/Discrete_sine_transform

[R45] 'Romberg's method' http://en.wikipedia.org/wiki/Romberg%27s_method

[R50] Wikipedia page: http://en.wikipedia.org/wiki/Trapezoidal_rule

[R51] Illustration image: http://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

[HNW93] E. Hairer, S.P. Norsett and G. Wanner, Solving Ordinary Differential Equations i. Nonstiff Problems. 2nd edition. Springer Series in Computational Mathematics, Springer-Verlag (1993)

[R46] J. Kierzenka, L. F. Shampine, "A BVP Solver Based on Residual Control and the Maltab PSE", ACM Trans. Math. Softw., Vol. 27, Number 3, pp. 299-316, 2001.

[R47] L.F. Shampine, P. H. Muir and H. Xu, "A User-Friendly Fortran BVP Solver".

[R48] U. Ascher, R. Mattheij and R. Russell "Numerical Solution of Boundary Value Problems for Ordinary Differential Equations".

[R49] Cauchy-Riemann equations on Wikipedia.

[R60] Krogh, "Efficient Algorithms for Polynomial Interpolation and Numerical Differentiation", 1970.

[R62] F. N. Fritsch and R. E. Carlson, Monotone Piecewise Cubic Interpolation, SIAM J. Numer. Anal., 17(2), 238 (1980). DOI:10.1137/0717021.

[R63] see, e.g., C. Moler, Numerical Computing with Matlab, 2004. DOI:10.1137/1.9780898717952

[R58] Cubic Spline Interpolation on Wikiversity.

[R59] Carl de Boor, "A Practical Guide to Splines", Springer-Verlag, 1978.

[R52] http://en.wikipedia.org/wiki/Bernstein_polynomial

[R53] Kenneth I. Joy, Bernstein polynomials, http://www.idav.ucdavis.edu/education/CAGDNotes/Bernstein-Polynomials.pdf

[R54] E. H. Doha, A. H. Bhrawy, and M. A. Saker, Boundary Value Problems, vol 2011, article ID 829546, DOI:10.1155/2011/829543.

[R61] http://www.qhull.org/

[R57] http://www.qhull.org/

[CT] See, for example, P. Alfeld, ''A trivariate Clough-Tocher scheme for tetrahedral data''. Computer Aided Geometric Design, 1, 169 (1984); G. Farin, ''Triangular Bernstein-Bezier patches''. Computer Aided Geometric Design, 3, 83 (1986).

[Nielson83] G. Nielson, ''A method for interpolating scattered data based upon a minimum norm network''. Math. Comp., 40, 253 (1983).

[Renka84] R. J. Renka and A. K. Cline. ''A Triangle-based C1 interpolation method.'', Rocky Mountain J. Math., 14, 223 (1984).

[R64] Python package *regulargrid* by Johannes Buchner, see https://pypi.python.org/pypi/regulargrid/

[R65] Trilinear interpolation. (2013, January 17). In Wikipedia, The Free Encyclopedia. Retrieved 27 Feb 2013 01:28. http://en.wikipedia.org/w/index.php?title=Trilinear_interpolation&oldid=533448871

[R66] Weiser, Alan, and Sergio E. Zarantonello. "A note on piecewise linear and multilinear table interpolation in many dimensions." MATH. COMPUT. 50.181 (1988): 189-196. http://www.ams.org/journals/mcom/1988-50-181/S0025-5718-1988-0917826-0/S0025-5718-1988-0917826-0.pdf

[R55] Tom Lyche and Knut Morken, Spline methods, http://www.uio.no/studier/emner/matnat/ifi/INF-MAT5340/v05/undervisningsmateriale/

[R56] Carl de Boor, A practical guide to splines, Springer, 2001.

[R86] P. Dierckx, "An algorithm for smoothing, differentiation and integration of experimental data using spline functions", J.Comp.Appl.Maths 1 (1975) 165-184.

[R87] P. Dierckx, "A fast algorithm for smoothing data on a rectangular grid while using spline functions", SIAM J.Numer.Anal. 19 (1982) 1286-1304.

[R88] P. Dierckx, "An improved algorithm for curve fitting with spline functions", report tw54, Dept. Computer Science,K.U. Leuven, 1981.

[R89] P. Dierckx, "Curve and surface fitting with splines", Monographs on Numerical Analysis, Oxford University Press, 1993.

[R83] P. Dierckx, "Algorithms for smoothing data with periodic and parametric splines, Computer Graphics and Image Processing", 20 (1982) 171-184.

[R84] P. Dierckx, "Algorithms for smoothing data with periodic and parametric splines", report tw55, Dept. Computer Science, K.U.Leuven, 1981.

[R85] P. Dierckx, "Curve and surface fitting with splines", Monographs on Numerical Analysis, Oxford University Press, 1993.

[R78] C. de Boor, "On calculating with b-splines", J. Approximation Theory, 6, p.50-62, 1972.

[R79] M. G. Cox, "The numerical evaluation of b-splines", J. Inst. Maths Applics, 10, p.134-149, 1972.

[R80] P. Dierckx, "Curve and surface fitting with splines", Monographs on Numerical Analysis, Oxford University Press, 1993.

[R81] P.W. Gaffney, The calculation of indefinite integrals of b-splines", J. Inst. Maths Applics, 17, p.37-41, 1976.

[R82] P. Dierckx, "Curve and surface fitting with splines", Monographs on Numerical Analysis, Oxford University Press, 1993.

[R90] C. de Boor, "On calculating with b-splines", J. Approximation Theory, 6, p.50-62, 1972.

[R91] M. G. Cox, "The numerical evaluation of b-splines", J. Inst. Maths Applics, 10, p.134-149, 1972.

[R92] P. Dierckx, "Curve and surface fitting with splines", Monographs on Numerical Analysis, Oxford University Press, 1993.

[R75] C. de Boor: On calculating with b-splines, J. Approximation Theory 6 (1972) 50-62.

[R76] M. G. Cox : The numerical evaluation of b-splines, J. Inst. Maths applics 10 (1972) 134-149.

[R77] P. Dierckx : Curve and surface fitting with splines, Monographs on Numerical Analysis, Oxford University Press, 1993.

[R73] W. Boehm, "Inserting new knots into b-spline curves.", Computer Aided Design, 12, p.199-201, 1980.

[R74] P. Dierckx, "Curve and surface fitting with splines, Monographs on Numerical Analysis", Oxford University Press, 1993.

[R70] Dierckx P.:An algorithm for surface fitting with spline functions Ima J. Numer. Anal. 1 (1981) 267-283.

[R71] Dierckx P.:An algorithm for surface fitting with spline functions report tw50, Dept. Computer Science,K.U.Leuven, 1980.

[R72] Dierckx P.:Curve and surface fitting with splines, Monographs on Numerical Analysis, Oxford University Press, 1993.

[R67] Dierckx P. : An algorithm for surface fitting with spline functions Ima J. Numer. Anal. 1 (1981) 267-283.

[R68] Dierckx P. : An algorithm for surface fitting with spline functions report tw50, Dept. Computer Science,K.U.Leuven, 1980.

[R69] Dierckx P. : Curve and surface fitting with splines, Monographs on Numerical Analysis, Oxford University Press, 1993.

[R93] IBM Corporation and Microsoft Corporation, "Multimedia Programming Interface and Data Specifications 1.0", section "Data Format of the Samples", August 1991 http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/Docs/riffmci.pdf

[R94] IBM Corporation and Microsoft Corporation, "Multimedia Programming Interface and Data Specifications 1.0", section "Data Format of the Samples", August 1991 http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/Docs/riffmci.pdf

[R111] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

[R112] Peter H. Schonemann, "A generalized solution of the orthogonal Procrustes problem", Psychometrica – Vol. 31, No. 1, March, 1996.

[R108] : B.N. Parlett and C. Reinsch, "Balancing a Matrix for Calculation of Eigenvalues and Eigenvectors", Numerische Mathematik, Vol.13(4), 1969, DOI:10.1007/BF02165404

[R109] : R. James, J. Langou, B.R. Lowery, "On matrix balancing and eigenvector computation", 2014, Available online: http://arxiv.org/abs/1401.5766

[R110] : D.S. Watkins. A case where balancing is harmful. Electron. Trans. Numer. Anal, Vol.23, 2006.

[R113] R. A. Horn and C. R. Johnson, "Matrix Analysis", Cambridge University Press, 1985.

[R114] N. J. Higham, "Functions of Matrices: Theory and Computation", SIAM, 2008.

[R121] Golub, G. H. & Van Loan, C. F. Matrix Computations, 3rd Ed. (Johns Hopkins University Press, 1996).

[R122] Daniel, J. W., Gragg, W. B., Kaufman, L. & Stewart, G. W. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. Math. Comput. 30, 772-795 (1976).

[R123] Reichel, L. & Gragg, W. B. Algorithm 686: FORTRAN Subroutines for Updating the QR Decomposition. ACM Trans. Math. Softw. 16, 369-377 (1990).

[R115] Golub, G. H. & Van Loan, C. F. Matrix Computations, 3rd Ed. (Johns Hopkins University Press, 1996).

[R116] Daniel, J. W., Gragg, W. B., Kaufman, L. & Stewart, G. W. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. Math. Comput. 30, 772-795 (1976).

[R117] Reichel, L. & Gragg, W. B. Algorithm 686: FORTRAN Subroutines for Updating the QR Decomposition. ACM Trans. Math. Softw. 16, 369-377 (1990).

[R118] Golub, G. H. & Van Loan, C. F. Matrix Computations, 3rd Ed. (Johns Hopkins University Press, 1996).

[R119] Daniel, J. W., Gragg, W. B., Kaufman, L. & Stewart, G. W. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. Math. Comput. 30, 772-795 (1976).

[R120] Reichel, L. & Gragg, W. B. Algorithm 686: FORTRAN Subroutines for Updating the QR Decomposition. ACM Trans. Math. Softw. 16, 369-377 (1990).

[R97] Awad H. Al-Mohy and Nicholas J. Higham (2009) "A New Scaling and Squaring Algorithm for the Matrix Exponential." SIAM Journal on Matrix Analysis and Applications. 31 (3). pp. 970-989. ISSN 1095-7162

[R105] Awad H. Al-Mohy and Nicholas J. Higham (2012) "Improved Inverse Scaling and Squaring Algorithms for the Matrix Logarithm." SIAM Journal on Scientific Computing, 34 (4). C152-C169. ISSN 1095-7197

[R106] Nicholas J. Higham (2008) "Functions of Matrices: Theory and Computation" ISBN 978-0-898716-46-7

[R107] Nicholas J. Higham and Lijing lin (2011) "A Schur-Pade Algorithm for Fractional Powers of a Matrix." SIAM Journal on Matrix Analysis and Applications, 32 (3). pp. 1056-1078. ISSN 0895-4798

[R132] Edvin Deadman, Nicholas J. Higham, Rui Ralha (2013) "Blocked Schur Algorithms for Computing the Matrix Square Root, Lecture Notes in Computer Science, 7782. pp. 171-182.

[R100] Gene H. Golub, Charles F. van Loan, Matrix Computations 4th ed.

[R98] Awad H. Al-Mohy and Nicholas J. Higham (2009) Computing the Frechet Derivative of the Matrix Exponential, with an application to Condition Number Estimation. SIAM Journal On Matrix Analysis and Applications., 30 (4). pp. 1639-1657. ISSN 1095-7162

[R99] Nicholas J. Higham and Lijing lin (2011) "A Schur-Pade Algorithm for Fractional Powers of a Matrix." SIAM Journal on Matrix Analysis and Applications, 32 (3). pp. 1056-1078. ISSN 0895-4798

[R124] P. van Dooren , "A Generalized Eigenvalue Approach For Solving Riccati Equations.", SIAM Journal on Scientific and Statistical Computing, Vol.2(2), DOI: 10.1137/0902010

[R125] A.J. Laub, "A Schur Method for Solving Algebraic Riccati Equations.", Massachusetts Institute of Technology. Laboratory for Information and Decision Systems. LIDS-R ; 859. Available online : http://hdl.handle.net/1721.1/1301

[R126] P. Benner, "Symplectic Balancing of Hamiltonian Matrices", 2001, SIAM J. Sci. Comput., 2001, Vol.22(5), DOI: 10.1137/S1064827500367993

[R127] P. van Dooren , "A Generalized Eigenvalue Approach For Solving Riccati Equations.", SIAM Journal on Scientific and Statistical Computing, Vol.2(2), DOI: 10.1137/0902010

[R128] A.J. Laub, "A Schur Method for Solving Algebraic Riccati Equations.", Massachusetts Institute of Technology. Laboratory for Information and Decision Systems. LIDS-R ; 859. Available online : http://hdl.handle.net/1721.1/1301

[R129] P. Benner, "Symplectic Balancing of Hamiltonian Matrices", 2001, SIAM J. Sci. Comput., 2001, Vol.22(5), DOI: 10.1137/S1064827500367993

[R130] Hamilton, James D. Time Series Analysis, Princeton: Princeton University Press, 1994. 265. Print. http://www.scribd.com/doc/20577138/Hamilton-1994-Time-Series-Analysis

[R131] Gajic, Z., and M.T.J. Qureshi. 2008. Lyapunov Matrix Equation in System Stability and Control. Dover Books on Engineering Series. Dover Publications.

[R95] R. A. Horn & C. R. Johnson, *Matrix Analysis*. Cambridge, UK: Cambridge University Press, 1999, pp. 146-7.

[R96] "DFT matrix", http://en.wikipedia.org/wiki/DFT_matrix

[R103] P. H. Leslie, On the use of matrices in certain population mathematics, Biometrika, Vol. 33, No. 3, 183–212 (Nov. 1945)

[R104] P. H. Leslie, Some further notes on the use of matrices in population mathematics, Biometrika, Vol. 35, No. 3/4, 213–245 (Dec. 1948)

[R101] "Pascal matrix", http://en.wikipedia.org/wiki/Pascal_matrix

[R102] Cohen, A. M., "The inverse of a Pascal matrix", Mathematical Gazette, 59(408), pp. 111-112, 1975.

[R742] P.G. Martinsson, V. Rokhlin, Y. Shkolnisky, M. Tygert. "ID: a software package for low-rank approximation of matrices via interpolative decompositions, version 0.2." http://cims.nyu.edu/~tygert/id_doc.pdf.

[R743] H. Cheng, Z. Gimbutas, P.G. Martinsson, V. Rokhlin. "On the compression of low rank matrices." *SIAM J. Sci. Comput.* 26 (4): 1389–1404, 2005. doi:10.1137/030602678.

[R744] E. Liberty, F. Woolfe, P.G. Martinsson, V. Rokhlin, M. Tygert. "Randomized algorithms for the low-rank approximation of matrices." *Proc. Natl. Acad. Sci. U.S.A.* 104 (51): 20167–20172, 2007. doi:10.1073/pnas.0709640104.

[R745] P.G. Martinsson, V. Rokhlin, M. Tygert. "A randomized algorithm for the decomposition of matrices." *Appl. Comput. Harmon. Anal.* 30 (1): 47–68, 2011. doi:10.1016/j.acha.2010.02.003.

[R746] F. Woolfe, E. Liberty, V. Rokhlin, M. Tygert. "A fast randomized algorithm for the approximation of matrices." *Appl. Comput. Harmon. Anal.* 25 (3): 335–366, 2008. doi:10.1016/j.acha.2007.12.002.

[R151] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2777

[R152] http://www.richardhartersworld.com/cri/2001/slidingmin.html

[R153] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2777

[R154] http://www.richardhartersworld.com/cri/2001/slidingmin.html

[R156] A.X. Falcao, J. Stolfi and R. de Alencar Lotufo, "The image foresting transform: theory, algorithms, and applications", Pattern Analysis and Machine Intelligence, vol. 26, pp. 19-29, 2004.

[R133] http://en.wikipedia.org/wiki/Closing_%28morphology%29

[R134] http://en.wikipedia.org/wiki/Mathematical_morphology

[R135] http://en.wikipedia.org/wiki/Dilation_%28morphology%29

[R136] http://en.wikipedia.org/wiki/Mathematical_morphology

[R137] http://en.wikipedia.org/wiki/Erosion_%28morphology%29

[R138] http://en.wikipedia.org/wiki/Mathematical_morphology

[R139] http://en.wikipedia.org/wiki/Mathematical_morphology

[R140] http://en.wikipedia.org/wiki/Hit-or-miss_transform

[R141] http://en.wikipedia.org/wiki/Opening_%28morphology%29

[R142] http://en.wikipedia.org/wiki/Mathematical_morphology

[R143] http://cmm.ensmp.fr/~serra/cours/pdf/en/ch6en.pdf, slide 15.

[R144] http://www.qi.tnw.tudelft.nl/Courses/FIP/noframes/fip-Morpholo.html#Heading102

[R145] http://en.wikipedia.org/wiki/Mathematical_morphology

[R146] http://en.wikipedia.org/wiki/Dilation_%28morphology%29

[R147] http://en.wikipedia.org/wiki/Mathematical_morphology

[R148] http://en.wikipedia.org/wiki/Erosion_%28morphology%29

[R149] http://en.wikipedia.org/wiki/Mathematical_morphology

[R150] http://en.wikipedia.org/wiki/Mathematical_morphology

[R155] http://en.wikipedia.org/wiki/Mathematical_morphology

[R771] P. T. Boggs and J. E. Rogers, "Orthogonal Distance Regression," in "Statistical analysis of measurement error models and applications: proceedings of the AMS-IMS-SIAM joint summer research conference held June 10-16, 1989," Contemporary Mathematics, vol. 112, pg. 186, 1990.

[R174] Nelder, J A, and R Mead. 1965. A Simplex Method for Function Minimization. The Computer Journal 7: 308-13.

[R175] Wright M H. 1996. Direct search methods: Once scorned, now respectable, in Numerical Analysis 1995: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis (Eds. D F Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK. 191-208.

[R176] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal 7: 155-162.

[R177] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.

[R178] Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.

[R179] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208.

[R180] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550-560.

[R181] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770-778.

[R182] Powell, M J D. A direct search optimization method that models the objective and constraint functions by linear interpolation. 1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), 51-67.

[R168] Nelder, J.A. and Mead, R. (1965), “A simplex method for function minimization”, The Computer Journal, 7, pp. 308-313

[R169] Wright, M.H. (1996), “Direct Search Methods: Once Scorned, Now Respectable”, in Numerical Analysis 1995, Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis, D.F. Griffiths and G.A. Watson (Eds.), Addison Wesley Longman, Harlow, UK, pp. 191-208.

[R170] Wright & Nocedal, “Numerical Optimization”, 1999, pp. 120-122.

[R164] Storn, R and Price, K, Differential Evolution - a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, 1997, 11, 341 - 359.

[R165] http://www1.icsi.berkeley.edu/~storn/code.html

[R166] http://en.wikipedia.org/wiki/Differential_evolution

[STIR] M. A. Branch, T. F. Coleman, and Y. Li, “A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems,” SIAM Journal on Scientific Computing, Vol. 21, Number 1, pp 1-23, 1999.

[NR] William H. Press et. al., “Numerical Recipes. The Art of Scientific Computing. 3rd edition”, Sec. 5.7.

[Byrd] R. H. Byrd, R. B. Schnabel and G. A. Shultz, “Approximate solution of the trust region problem by minimization over two-dimensional subspaces”, Math. Programming, 40, pp. 247-263, 1988.

[Curtis] A. Curtis, M. J. D. Powell, and J. Reid, “On the estimation of sparse Jacobian matrices”, Journal of the Institute of Mathematics and its Applications, 13, pp. 117-120, 1974.

[JJMore] J. J. More, “The Levenberg-Marquardt Algorithm: Implementation and Theory,” Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag, pp. 105-116, 1977.

[Voglis] C. Voglis and I. E. Lagaris, “A Rectangular Trust Region Dogleg Approach for Unconstrained and Bound Constrained Nonlinear Optimization”, WSEAS International Conference on Applied Mathematics, Corfu, Greece, 2004.

[NumOpt] J. Nocedal and S. J. Wright, “Numerical optimization, 2nd edition”, Chapter 4.

[BA] B. Triggs et. al., “Bundle Adjustment - A Modern Synthesis”, Proceedings of the International Workshop on Vision Algorithms: Theory and Practice, pp. 298-372, 1999.

[STIR] M. A. Branch, T. F. Coleman, and Y. Li, "A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems," SIAM Journal on Scientific Computing, Vol. 21, Number 1, pp 1-23, 1999.

[BVLS] P. B. Start and R. L. Parker, "Bounded-Variable Least-Squares: an Algorithm and Applications", Computational Statistics, 10, 129-141, 1995.

[R158] Wales, David J. 2003, Energy Landscapes, Cambridge University Press, Cambridge, UK.

[R159] Wales, D J, and Doye J P K, Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. Journal of Physical Chemistry A, 1997, 101, 5111.

[R160] Li, Z. and Scheraga, H. A., Monte Carlo-minimization approach to the multiple-minima problem in protein folding, Proc. Natl. Acad. Sci. USA, 1987, 84, 6611.

[R161] Wales, D. J. and Scheraga, H. A., Global optimization of clusters, crystals, and biomolecules, Science, 1999, 285, 1368.

[Brent1973] Brent, R. P., *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973. Ch. 3-4.

[PressEtal1992] Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press, pp. 352-355, 1992. Section 9.3: "Van Wijngaarden-Dekker-Brent Method."

[Ridders1979] Ridders, C. F. J. "A New Algorithm for Computing a Single Root of a Real Continuous Function." IEEE Trans. Circuits Systems 26, 979-980, 1979.

[R167] Burden, Faires, "Numerical Analysis", 5th edition, pg. 80

[R185] More, Jorge J., Burton S. Garbow, and Kenneth E. Hillstrom. 1980. User Guide for MINPACK-1.

[R186] C. T. Kelley. 1995. Iterative Methods for Linear and Nonlinear Equations. Society for Industrial and Applied Mathematics. <http://www.siam.org/books/kelley/>

[R187] 23. La Cruz, J.M. Martinez, M. Raydan. Math. Comp. 75, 1429 (2006).

[R162] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003).

http://www.math.leidenuniv.nl/scripties/Rotten.pdf

[R163] B.A. van der Rotten, PhD thesis, "A limited memory Broyden method to solve high-dimensional systems of nonlinear equations". Mathematisch Instituut, Universiteit Leiden, The Netherlands (2003).

http://www.math.leidenuniv.nl/scripties/Rotten.pdf

[R813] "Spectral residual method without gradient information for solving large-scale nonlinear systems of equations." W. La Cruz, J.M. Martinez, M. Raydan. Math. Comp. **75**, 1429 (2006).

[R814] 23. La Cruz, Opt. Meth. Software, 29, 24 (2014).

[R815] 23. Cheng, D.-H. Li. IMA J. Numer. Anal. **29**, 814 (2009).

[R183] D.A. Knoll and D.E. Keyes, J. Comp. Phys. 193, 357 (2004). DOI:10.1016/j.jcp.2003.08.010

[R184] A.H. Baker and E.R. Jessup and T. Manteuffel, SIAM J. Matrix Anal. Appl. 26, 962 (2005). DOI:10.1137/S0895479803422014

[Ey] 22. Eyert, J. Comp. Phys., 124, 271 (1996).

[R171] Dantzig, George B., Linear programming and extensions. Rand Corporation Research Study Princeton Univ. Press, Princeton, NJ, 1963

[R172] Hillier, S.H. and Lieberman, G.J. (1995), "Introduction to Mathematical Programming", McGraw-Hill, Chapter 4.

[R173] Bland, Robert G. New finite pivoting rules for the simplex method. Mathematics of Operations Research (2), 1977: pp. 103-107.

[R806] Dantzig, George B., Linear programming and extensions. Rand Corporation Research Study Princeton Univ. Press, Princeton, NJ, 1963

[R807] Hillier, S.H. and Lieberman, G.J. (1995), "Introduction to Mathematical Programming", McGraw-Hill, Chapter 4.

[R808] Bland, Robert G. New finite pivoting rules for the simplex method. Mathematics of Operations Research (2), 1977: pp. 103-107.

[R157] Nocedal, Jorge. "Updating quasi-Newton matrices with limited storage." Mathematics of computation 35.151 (1980): 773-782.

[R216] F. Gustaffson, "Determining the initial states in forward-backward filtering", Transactions on Signal Processing, Vol. 46, pp. 988-992, 1996.

[R235] Wikipedia, "Analytic signal". http://en.wikipedia.org/wiki/Analytic_signal

[R236] Leon Cohen, "Time-Frequency Analysis", 1995. Chapter 2.

[R237] Alan V. Oppenheim, Ronald W. Schafer. Discrete-Time Signal Processing, Third Edition, 2009. Chapter 12. ISBN 13: 978-1292-02572-8

[R238] Wikipedia, "Analytic signal", http://en.wikipedia.org/wiki/Analytic_signal

[R269] P. P. Vaidyanathan, Multirate Systems and Filter Banks, Prentice Hall, 1993.

[R218] Ivan Selesnick, Linear-Phase Fir Filter Design By Least Squares. OpenStax CNX. Aug 9, 2005. http://cnx.org/contents/eb1ecb35-03a9-4610-ba87-41cd771c95f2@7

[R219] Oppenheim, A. V. and Schafer, R. W., "Discrete-Time Signal Processing", Prentice-Hall, Englewood Cliffs, New Jersey (1989). (See, for example, Section 7.4.)

[R220] Smith, Steven W., "The Scientist and Engineer's Guide to Digital Signal Processing", Ch. 17. http://www.dspguide.com/ch17/1.htm

[R222] Richard G. Lyons, "Understanding Digital Signal Processing, 3rd edition", p. 830.

[R250] N. Damera-Venkata and B. L. Evans, "Optimal design of real and complex minimum phase digital FIR filters," Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on, Phoenix, AZ, 1999, pp. 1145-1148 vol.3. doi: 10.1109/ICASSP.1999.756179

[R251] X. Chen and T. W. Parks, "Design of optimal minimum phase FIR filters by direct factorization," Signal Processing, vol. 10, no. 4, pp. 369–383, Jun. 1986.

[R252] T. Saramaki, "Finite Impulse Response Filter Design," in Handbook for Digital Signal Processing, chapter 4, New York: Wiley-Interscience, 1993.

[R253] J. S. Lim, Advanced Topics in Signal Processing. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[R254] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, "Discrete-Time Signal Processing," 2nd edition. Upper Saddle River, N.J.: Prentice Hall, 1999.

[R260] J. H. McClellan and T. W. Parks, "A unified approach to the design of optimum FIR linear phase digital filters", IEEE Trans. Circuit Theory, vol. CT-20, pp. 697-701, 1973.

[R261] J. H. McClellan, T. W. Parks and L. R. Rabiner, "A Computer Program for Designing Optimum FIR Linear Phase Digital Filters", IEEE Trans. Audio Electroacoust., vol. AU-21, pp. 506-525, 1973.

[R194] C.R. Bond, "Bessel Filter Constants", http://www.crbond.com/papers/bsf.pdf

[R195] Campos and Calderon, "Approximate closed-form formulas for the zeros of the Bessel Polynomials", arXiv:1105.0957.

[R196] Thomson, W.E., "Delay Networks having Maximally Flat Frequency Characteristics", Proceedings of the Institution of Electrical Engineers, Part III, November 1949, Vol. 96, No. 44, pp. 487-490.

[R197] Aberth, "Iteration Methods for Finding all Zeros of a Polynomial Simultaneously", Mathematics of Computation, Vol. 27, No. 122, April 1973

[R198] Ehrlich, "A modified Newton method for polynomials", Communications of the ACM, Vol. 10, Issue 2, pp. 107-108, Feb. 1967, DOI:10.1145/363067.363115

[R199] Miller and Bohn, "A Bessel Filter Crossover, and Its Relation to Others", RaneNote 147, 1998, http://www.rane.com/note147.html

[R215] Lutova, Tosic, and Evans, "Filter Design for Signal Processing", Chapters 5 and 12.

[R193] Thomson, W.E., "Delay Networks having Maximally Flat Frequency Characteristics", Proceedings of the Institution of Electrical Engineers, Part III, November 1949, Vol. 96, No. 44, pp. 487-490.

[R239] Sophocles J. Orfanidis, "Introduction To Signal Processing", Prentice-Hall, 1996

[R240] Sophocles J. Orfanidis, "Introduction To Signal Processing", Prentice-Hall, 1996

[R210] http://en.wikipedia.org/wiki/Discretization#Discretization_of_linear_state_space_models

[R211] http://techteach.no/publications/discretetime_signals_systems/discrete.pdf

[R212] G. Zhang, X. Chen, and T. Chen, Digital redesign via the generalized bilinear transformation, Int. J. Control, vol. 82, no. 4, pp. 741-754, 2009. (http://www.ece.ualberta.ca/~gfzhang/research/ZCC07_preprint.pdf)

[R258] J. Kautsky, N.K. Nichols and P. van Dooren, "Robust pole assignment in linear state feedback", International Journal of Control, Vol. 41 pp. 1129-1155, 1985.

[R259] A.L. Tits and Y. Yang, "Globally convergent algorithms for robust pole assignment by state feedback, IEEE Transactions on Automatic Control, Vol. 41, pp. 1432-1452, 1996.

[R188] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra", Biometrika 37, 1-16, 1950.

[R189] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.

[R190] A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal Processing", Prentice-Hall, 1999, pp. 468-471.

[R191] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

[R192] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 429.

[R200] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[R201] Oppenheim, A.V., and R.W. Schafer. Discrete-Time Signal Processing. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

[R202] Harris, Fredric J. (Jan 1978). "On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform". Proceedings of the IEEE 66 (1): 51-83. DOI:10.1109/PROC.1978.10837.

[R203] C. Dolph, "A current distribution for broadside arrays which optimizes the relationship between beam width and side-lobe level", Proceedings of the IEEE, Vol. 34, Issue 6

[R204] Peter Lynch, "The Dolph-Chebyshev Window: A Simple Optimal Filter", American Meteorological Society (April 1997) http://mathsci.ucd.ie/~plynch/Publications/Dolph.pdf

[R205] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transforms", Proceedings of the IEEE, Vol. 66, No. 1, January 1978

[R221] D'Antona, Gabriele, and A. Ferrero, "Digital Signal Processing for Measurement Systems", Springer Media, 2006, p. 70 DOI:10.1007/0-387-28666-7.

[R223] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[R224] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 109-110.

[R225] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

[R226] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

[R227] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[R228] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 106-108.

[R229] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

[R230] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

[R231] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[R232] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 106-108.

[R233] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

[R234] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes", Cambridge University Press, 1986, page 425.

[R243] J. F. Kaiser, "Digital Filters" - Ch 7 in "Systems analysis by digital computer", Editors: F.F. Kuo and J.F. Kaiser, p 218-285. John Wiley and Sons, New York, (1966).

[R244] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The University of Alberta Press, 1975, pp. 177-178.

[R245] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function

[R246] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," Proceedings of the IEEE, vol. 66, no. 1, pp. 51-83, Jan. 1978. DOI:10.1109/PROC.1978.10837.

[R255] A. Nuttall, "Some windows with very good sidelobe behavior," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 29, no. 1, pp. 84-91, Feb 1981. DOI:10.1109/TASSP.1981.1163506.

[R256] Heinzel G. et al., "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows", February 15, 2002 https://holometer.fnal.gov/GH_FFT.pdf

[R257] E. Parzen, "Mathematical Considerations in the Estimation of Spectra", Technometrics, Vol. 3, No. 2 (May, 1961), pp. 167-190

[R262] D. Slepian & H. O. Pollak: "Prolate spheroidal wave functions, Fourier analysis and uncertainty-I," Bell Syst. Tech. J., vol.40, pp.43-63, 1961. https://archive.org/details/bstj40-1-43

[R263] H. J. Landau & H. O. Pollak: "Prolate spheroidal wave functions, Fourier analysis and uncertainty-II," Bell Syst. Tech. J. , vol.40, pp.65-83, 1961. https://archive.org/details/bstj40-1-65

[R267] Harris, Fredric J. (Jan 1978). "On the use of Windows for Harmonic Analysis with the Discrete Fourier Transform". Proceedings of the IEEE 66 (1): 51-83. DOI:10.1109/PROC.1978.10837

[R268] Wikipedia, "Window function", http://en.wikipedia.org/wiki/Window_function#Tukey_window

[R217] Bioinformatics (2006) 22 (17): 2059-2065. DOI:10.1093/bioinformatics/btl355 http://bioinformatics.oxfordjournals.org/content/22/17/2059.long

[R270] P. Welch, “The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”, IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

[R271] M.S. Bartlett, “Periodogram Analysis and Continuous Spectra”, Biometrika, vol. 37, pp. 1-16, 1950.

[R213] P. Welch, “The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”, IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

[R214] Rabiner, Lawrence R., and B. Gold. “Theory and Application of Digital Signal Processing” Prentice-Hall, pp. 414-419, 1975

[R208] P. Welch, “The use of the fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”, IEEE Trans. Audio Electroacoust. vol. 15, pp. 70-73, 1967.

[R209] Stoica, Petre, and Randolph Moses, “Spectral Analysis of Signals” Prentice Hall, 2005

[R264] Oppenheim, Alan V., Ronald W. Schafer, John R. Buck “Discrete-Time Signal Processing”, Prentice Hall, 1999.

[R247] N.R. Lomb “Least-squares frequency analysis of unequally spaced data”, Astrophysics and Space Science, vol 39, pp. 447-462, 1976

[R248] J.D. Scargle “Studies in astronomical time series analysis. II - Statistical aspects of spectral analysis of unevenly spaced data”, The Astrophysical Journal, vol 263, pp. 835-853, 1982

[R249] R.H.D. Townsend, “Fast calculation of the Lomb-Scargle periodogram using graphics processing units.”, The Astrophysical Journal Supplement Series, vol 191, pp. 247-253, 2010

[R265] Oppenheim, Alan V., Ronald W. Schafer, John R. Buck “Discrete-Time Signal Processing”, Prentice Hall, 1999.

[R266] Daniel W. Griffin, Jae S. Limdt “Signal Estimation from Modified Short Fourier Transform”, IEEE 1984, 10.1109/TASSP.1984.1164317

[R241] Oppenheim, Alan V., Ronald W. Schafer, John R. Buck “Discrete-Time Signal Processing”, Prentice Hall, 1999.

[R242] Daniel W. Griffin, Jae S. Limdt “Signal Estimation from Modified Short Fourier Transform”, IEEE 1984, 10.1109/TASSP.1984.1164317

[R206] Julius O. Smith III, “Spectral Audio Signal Processing”, W3K Publishing, 2011,ISBN 978-0-9745607-3-1.

[R207] G. Heinzel, A. Ruediger and R. Schilling, “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new at-top windows”, 2002, http://hdl.handle.net/11858/00-001M-0000-0013-557A-5

[R14] D. J. Pearce, “An Improved Algorithm for Finding the Strongly Connected Components of a Directed Graph”, Technical Report, 2005

[R15] I. S. Duff, “Computing the Structural Index”, SIAM J. Alg. Disc. Meth., Vol. 7, 594 (1986).

[R16] http://www.cise.ufl.edu/research/sparse/matrices/legend.html

[R21] Awad H. Al-Mohy and Nicholas J. Higham (2009) “A New Scaling and Squaring Algorithm for the Matrix Exponential.” SIAM Journal on Matrix Analysis and Applications. 31 (3). pp. 970-989. ISSN 1095-7162

[R22] Awad H. Al-Mohy and Nicholas J. Higham (2011) “Computing the Action of the Matrix Exponential, with an Application to Exponential Integrators.” SIAM Journal on Scientific Computing, 33 (2). pp. 488-511. ISSN 1064-8275 http://eprints.ma.man.ac.uk/1591/

[R23] Nicholas J. Higham and Awad H. Al-Mohy (2010) “Computing Matrix Functions.” Acta Numerica, 19. 159-208. ISSN 0962-4929 http://eprints.ma.man.ac.uk/1451/

[R34] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

[R35] Nicholas J. Higham and Francoise Tisseur (2000), "A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra." SIAM J. Matrix Anal. Appl. Vol. 21, No. 4, pp. 1185-1201.

[R36] Awad H. Al-Mohy and Nicholas J. Higham (2009), "A new scaling and squaring algorithm for the matrix exponential." SIAM J. Matrix Anal. Appl. Vol. 31, No. 3, pp. 970-989.

[R24] A.H. Baker and E.R. Jessup and T. Manteuffel, SIAM J. Matrix Anal. Appl. 26, 962 (2005).

[R25] A.H. Baker, PhD thesis, University of Colorado (2003). http://amath.colorado.edu/activities/thesis/allisonb/Thesis.ps

[R31] C. C. Paige and M. A. Saunders (1982a). "LSQR: An algorithm for sparse linear equations and sparse least squares", ACM TOMS 8(1), 43-71.

[R32] C. C. Paige and M. A. Saunders (1982b). "Algorithm 583. LSQR: Sparse linear equations and least squares problems", ACM TOMS 8(2), 195-209.

[R33] M. A. Saunders (1995). "Solution of sparse rectangular systems using LSQR and CRAIG", BIT 35, 588-604.

[R29] D. C.-L. Fong and M. A. Saunders, "LSMR: An iterative algorithm for sparse least-squares problems", SIAM J. Sci. Comput., vol. 33, pp. 2950-2971, 2011. http://arxiv.org/abs/1006.0758

[R30] LSMR Software, http://web.stanford.edu/group/SOL/software/lsmr/

[R17] ARPACK Software, http://www.caam.rice.edu/software/ARPACK/

[R18] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. SIAM, Philadelphia, PA, 1998.

[R19] ARPACK Software, http://www.caam.rice.edu/software/ARPACK/

[R20] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. SIAM, Philadelphia, PA, 1998.

[R26] A. V. Knyazev (2001), Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM Journal on Scientific Computing 23, no. 2, pp. 517-541. http://dx.doi.org/10.1137/S1064827500366124

[R27] A. V. Knyazev, I. Lashuk, M. E. Argentati, and E. Ovchinnikov (2007), Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in hypre and PETSc. http://arxiv.org/abs/0705.2626

[R28] A. V. Knyazev's C and MATLAB implementations: http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/

[R39] SuperLU http://crd.lbl.gov/~xiaoye/SuperLU/

[R331] Awad H. Al-Mohy and Nicholas J. Higham (2009) "A New Scaling and Squaring Algorithm for the Matrix Exponential." SIAM Journal on Matrix Analysis and Applications. 31 (3). pp. 970-989. ISSN 1095-7162

[R332] Awad H. Al-Mohy and Nicholas J. Higham (2011) "Computing the Action of the Matrix Exponential, with an Application to Exponential Integrators." SIAM Journal on Scientific Computing, 33 (2). pp. 488-511. ISSN 1064-8275 http://eprints.ma.man.ac.uk/1591/

[R333] Nicholas J. Higham and Awad H. Al-Mohy (2010) "Computing Matrix Functions." Acta Numerica, 19. 159-208. ISSN 0962-4929 http://eprints.ma.man.ac.uk/1451/

[R344] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

[R345] Nicholas J. Higham and Francoise Tisseur (2000), "A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra." SIAM J. Matrix Anal. Appl. Vol. 21, No. 4, pp. 1185-1201.

[R346] Awad H. Al-Mohy and Nicholas J. Higham (2009), "A new scaling and squaring algorithm for the matrix exponential." SIAM J. Matrix Anal. Appl. Vol. 31, No. 3, pp. 970-989.

[R334] A.H. Baker and E.R. Jessup and T. Manteuffel, SIAM J. Matrix Anal. Appl. 26, 962 (2005).

[R335] A.H. Baker, PhD thesis, University of Colorado (2003). http://amath.colorado.edu/activities/thesis/allisonb/Thesis.ps

[R341] C. C. Paige and M. A. Saunders (1982a). "LSQR: An algorithm for sparse linear equations and sparse least squares", ACM TOMS 8(1), 43-71.

[R342] C. C. Paige and M. A. Saunders (1982b). "Algorithm 583. LSQR: Sparse linear equations and least squares problems", ACM TOMS 8(2), 195-209.

[R343] M. A. Saunders (1995). "Solution of sparse rectangular systems using LSQR and CRAIG", BIT 35, 588-604.

[R339] D. C.-L. Fong and M. A. Saunders, "LSMR: An iterative algorithm for sparse least-squares problems", SIAM J. Sci. Comput., vol. 33, pp. 2950-2971, 2011. http://arxiv.org/abs/1006.0758

[R340] LSMR Software, http://web.stanford.edu/group/SOL/software/lsmr/

[R327] ARPACK Software, http://www.caam.rice.edu/software/ARPACK/

[R328] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. SIAM, Philadelphia, PA, 1998.

[R329] ARPACK Software, http://www.caam.rice.edu/software/ARPACK/

[R330] R. B. Lehoucq, D. C. Sorensen, and C. Yang, ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods. SIAM, Philadelphia, PA, 1998.

[R336] A. V. Knyazev (2001), Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM Journal on Scientific Computing 23, no. 2, pp. 517-541. http://dx.doi.org/10.1137/S1064827500366124

[R337] A. V. Knyazev, I. Lashuk, M. E. Argentati, and E. Ovchinnikov (2007), Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in hypre and PETSc. http://arxiv.org/abs/0705.2626

[R338] A. V. Knyazev's C and MATLAB implementations: http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/

[R349] SuperLU http://crd.lbl.gov/~xiaoye/SuperLU/

[R278] D. J. Pearce, "An Improved Algorithm for Finding the Strongly Connected Components of a Directed Graph", Technical Report, 2005

[R279] I. S. Duff, "Computing the Structural Index", SIAM J. Alg. Disc. Meth., Vol. 7, 594 (1986).

[R280] http://www.cise.ufl.edu/research/sparse/matrices/legend.html

[R356] Gray and Moore, "N-body problems in statistical learning", Mining the sky, 2000, https://arxiv.org/abs/astro-ph/0012333

[R357] Landy and Szalay, "Bias and variance of angular correlation functions", The Astrophysical Journal, 1993, http://adsabs.harvard.edu/abs/1993ApJ...412...64L

[R358] Sheth, Connolly and Skibba, "Marked correlations in galaxy formation models", Arxiv e-print, 2005, https://arxiv.org/abs/astro-ph/0511773

[R359] Hawkins, et al., "The 2dF Galaxy Redshift Survey: correlation functions, peculiar velocities and the matter density of the Universe", Monthly Notices of the Royal Astronomical Society, 2002, http://adsabs.harvard.edu/abs/2003MNRAS.346...78H

[R360] https://github.com/scipy/scipy/pull/5647#issuecomment-168474926

[R40] A. A. Taha and A. Hanbury, "An efficient algorithm for calculating the exact Hausdorff distance." IEEE Transactions On Pattern Analysis And Machine Intelligence, vol. 37 pp. 2153-63, 2015.

[Qhull] http://www.qhull.org/

[Caroli] Caroli et al. Robust and Efficient Delaunay triangulations of points on or close to a sphere. Research Report RR-7004, 2009.

[Weisstein] "L'Huilier's Theorem." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/LHuiliersTheorem.html

[Qhull] http://www.qhull.org/

[R350] S. Boyd, L. Vandenberghe, Convex Optimization, available at http://stanford.edu/~boyd/cvxbook/

[R365] Krzanowski, W. J. (2000). "Principles of Multivariate analysis".

[R366] Gower, J. C. (1975). "Generalized procrustes analysis".

[R364] A. A. Taha and A. Hanbury, "An efficient algorithm for calculating the exact Hausdorff distance." IEEE Transactions On Pattern Analysis And Machine Intelligence, vol. 37 pp. 2153-63, 2015.

[R368] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R369] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/.org/amos/

[R370] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R367] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R383] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R444] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R404] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R406] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R405] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R402] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R403] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R459] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R454] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R460] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R540] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R543] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R544] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R471] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R472] Donald E. Amos, "Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order", ACM TOMS Vol. 12 Issue 3, Sept. 1986, p. 265

[R473] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R474] Donald E. Amos, "Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order", ACM TOMS Vol. 12 Issue 3, Sept. 1986, p. 265

[R475] NIST Digital Library of Mathematical Functions, Eq. 10.25.E3. http://dlmf.nist.gov/10.25.E3

[R476] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R477] Donald E. Amos, "Algorithm 644: A portable package for Bessel functions of a complex argument and nonnegative order", ACM TOMS Vol. 12 Issue 3, Sept. 1986, p. 265

[R447] Temme, Journal of Computational Physics, vol 21, 343 (1976)

[R448] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R449] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R435] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R436] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R437] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R438] Donald E. Amos, "AMOS, A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order", http://netlib.org/amos/

[R482] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R483] Jahnke, E. and Emde, F. "Tables of Functions with Formulae and Curves" (4th ed.), Dover, 1945

[R456] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R458] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R455] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R457] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R541] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R542] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R536] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R538] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R539] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R452] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R453] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R535] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R537] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R439] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R440] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R441] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R442] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R463] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R464] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R465] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R466] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R461] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R462] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.6.E7

[R545] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R546] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.6.E7

[R478] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 6. http://jin.ece.illinois.edu/specfunc.html

[R479] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.29.E5

[R450] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 6. http://jin.ece.illinois.edu/specfunc.html

[R451] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.29.E5

[R431] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R432] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.6.E7

[R433] Zhang, Shanjie and Jin, Jianming. “Computation of Special Functions”, John Wiley and Sons, 1996, chapter 5. http://jin.ece.illinois.edu/specfunc.html

[R434] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.6.E7

[R524] http://dlmf.nist.gov/10.47.E3

[R525] http://dlmf.nist.gov/10.51.E1

[R526] http://dlmf.nist.gov/10.51.E2

[R529] http://dlmf.nist.gov/10.47.E4

[R530] http://dlmf.nist.gov/10.51.E1

[R531] http://dlmf.nist.gov/10.51.E2

[R522] http://dlmf.nist.gov/10.47.E7

[R523] http://dlmf.nist.gov/10.51.E5

[R527] http://dlmf.nist.gov/10.47.E9

[R528] http://dlmf.nist.gov/10.51.E5

[R510] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R511] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.51.E1

[R512] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R513] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/10.51.E1

[R532] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/11

[R493] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/11

[R446] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R443] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R445] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R371] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R372] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R373] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R374] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972.

[R375] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R376] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972.

[R377] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R384] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R385] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R386] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R387] DiDinato, A. R. and Morris, A. H., Algorithm 708: Significant Digit Computation of the Incomplete Beta Function Ratios. ACM Trans. Math. Softw. 18 (1993), 360-373.

[R388] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R389] DiDinato, A. R. and Morris, A. H., Algorithm 708: Significant Digit Computation of the Incomplete Beta Function Ratios. ACM Trans. Math. Softw. 18 (1993), 360-373.

[R415] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R416] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R417] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R423] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R424] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R425] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R426] DiDinato, A. R. and Morris, A. H., Computation of the incomplete gamma function ratios and their inverse. ACM Trans. Math. Softw. 12 (1986), 377-393.

[R427] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R428] DiDinato, A. R. and Morris, A. H., Computation of the incomplete gamma function ratios and their inverse. ACM Trans. Math. Softw. 12 (1986), 377-393.

[R429] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R430] DiDinato, A. R. and Morris, A. H., Computation of the incomplete gamma function ratios and their inverse. ACM Trans. Math. Softw. 12 (1986), 377-393.

[R494] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R495] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R496] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R497] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R498] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972.

[R499] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R500] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972.

[R501] Barry Brown, James Lovato, and Kathy Russell, CDFLIB: Library of Fortran Routines for Cumulative Distribution Functions, Inverses, and Other Parameters.

[R502] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972.

[hare1997] D.E.G. Hare, *Computing the Principal Branch of log-Gamma*, Journal of Algorithms, Volume 25, Issue 2, November 1997, pages 221-236.

[R421] Maddock et. al., "Incomplete Gamma Functions", http://www.boost.org/doc/libs/1_61_0/libs/math/doc/html/math_toolkit/sf_gamma/igamma.html

[R422] Maddock et. al., "Incomplete Gamma Functions", http://www.boost.org/doc/libs/1_61_0/libs/math/doc/html/math_toolkit/sf_gamma/igamma.html

[R508] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/5

[R509] Fredrik Johansson and others. "mpmath: a Python library for arbitrary-precision floating-point arithmetic" (Version 0.19) http://mpmath.org/

[R396] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/5

[R397] Fredrik Johansson and others. "mpmath: a Python library for arbitrary-precision floating-point arithmetic" (Version 0.19) http://mpmath.org/

[R407] http://en.wikipedia.org/wiki/Error_function

[R408] Milton Abramowitz and Irene A. Stegun, eds. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover, 1972. http://www.math.sfu.ca/~cbm/aands/page_297.htm

[R409] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R411] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R412] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R413] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R533] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R395] Steven G. Johnson, Faddeeva W function implementation. http://ab-initio.mit.edu/Faddeeva

[R418] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R410] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R419] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R420] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R486] Zhang, Jin, "Computation of Special Functions", John Wiley and Sons, Inc, 1996.

[R520] Digital Library of Mathematical Functions, 14.30. http://dlmf.nist.gov/14.30

[R521] https://en.wikipedia.org/wiki/Spherical_harmonics#Condon.E2.80.93Shortley_phase

[R392] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R393] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/14.21

[R487] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R489] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R484] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R485] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/14.3

[R488] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R398] Digital Libary of Mathematical Functions 29.12 http://dlmf.nist.gov/29.12

[R399] Bardhan and Knepley, "Computational science and re-discovery: open-source implementations of ellipsoidal harmonics for problems in potential theory", Comput. Sci. Disc. 5, 014006 (2012) DOI:10.1088/1749-4699/5/1/014006.

[R400] David J.and Dechambre P, "Computation of Ellipsoidal Gravity Field Harmonics for small solar system bodies" pp. 30-36, 2000

[R401] George Dassios, "Ellipsoidal Harmonics: Theory and Applications" pp. 418, 2012

[townsend.trogdon.olver-2014] Townsend, A. and Trogdon, T. and Olver, S. (2014) *Fast computation of Gauss quadrature nodes and weights on the whole real line*. arXiv:1410.5286.

[townsend.trogdon.olver-2015] Townsend, A. and Trogdon, T. and Olver, S. (2015) *Fast computation of Gauss quadrature nodes and weights on the whole real line*. IMA Journal of Numerical Analysis DOI:10.1093/imanum/drv002.

[R390] Abramowitz and Stegun, "Handbook of Mathematical Functions" Section 22. National Bureau of Standards, 1972.

[R391] Abramowitz and Stegun, "Handbook of Mathematical Functions" Section 22. National Bureau of Standards, 1972.

[R505] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 13. http://jin.ece.illinois.edu/specfunc.html

[R506] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 13. http://jin.ece.illinois.edu/specfunc.html

[R504] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996, chapter 13. http://jin.ece.illinois.edu/specfunc.html

[R490] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R491] NIST Digital Library of Mathematical Functions http://dlmf.nist.gov/28.4#i

[R492] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R507] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R503] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R468] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R380] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R378] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R382] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R379] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R469] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R470] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R467] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R381] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R414] Zhang, Shanjie and Jin, Jianming. "Computation of Special Functions", John Wiley and Sons, 1996. http://jin.ece.illinois.edu/specfunc.html

[R514] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R515] Fredrik Johansson and others. “mpmath: a Python library for arbitrary-precision floating-point arithmetic” (Version 0.19) http://mpmath.org/

[R516] Cephes Mathematical Functions Library, http://www.netlib.org/cephes/index.html

[R517] Fredrik Johansson and others. “mpmath: a Python library for arbitrary-precision floating-point arithmetic” (Version 0.19) http://mpmath.org/

[R518] Weisstein, Eric W. “Sinc Function.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/SincFunction.html

[R519] Wikipedia, “Sinc function”, http://en.wikipedia.org/wiki/Sinc_function

[R555] “ARGUS distribution”, https://en.wikipedia.org/wiki/ARGUS_distribution

[R561] Burr, I. W. “Cumulative frequency functions”, Annals of Mathematical Statistics, 13(2), pp 215-232 (1942).

[R562] Burr, I. W. “Cumulative frequency functions”, Annals of Mathematical Statistics, 13(2), pp 215-232 (1942).

[R563] http://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/b12pdf.htm

[R577] “Birnbaum-Saunders distribution”, http://en.wikipedia.org/wiki/Birnbaum-Saunders_distribution

[R587] “Generalized normal distribution, Version 1”, https://en.wikipedia.org/wiki/Generalized_normal_distribution#Version_1

[R588] “Generalized normal distribution, Version 1”, https://en.wikipedia.org/wiki/Generalized_normal_distribution#Version_1

[R606] http://mathworld.wolfram.com/MaxwellDistribution.html

[R636] A. Azzalini and A. Capitanio (1999). Statistical applications of the multivariate skew-normal distribution. J. Roy. Statist. Soc., B 61, 579-602. http://azzalini.stat.unipd.it/SN/faq-r.html

[R652] M.L. Eaton, “Multivariate Statistics: A Vector Space Approach”, Wiley, 1983.

[R653] W.B. Smith and R.R. Hocking, “Algorithm AS 53: Wishart Variate Generator”, Applied Statistics, vol. 21, pp. 341-345, 1972.

[R589] M.L. Eaton, “Multivariate Statistics: A Vector Space Approach”, Wiley, 1983.

[R590] M.C. Jones, “Generating Inverse Wishart Matrices”, Communications in Statistics - Simulation and Computation, vol. 14.2, pp.511-514, 1985.

[R617] F. Mezzadri, “How to generate random matrices from the classical compact groups”, arXiv:math-ph/0609050v2.

[R628] Davies, Philip I; Higham, Nicholas J; “Numerically stable generation of correlation matrices and their factors”, BIT 2000, Vol. 40, No. 4, pp. 640 651

[R601] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000.

[R602] see e.g. F. J. Anscombe, W. J. Glynn, “Distribution of the kurtosis statistic b2 for normal samples”, Biometrika, vol. 70, pp. 227-234, 1983.

[R609] http://eli.thegreenplace.net/2009/03/21/efficient-integer-exponentiation-algorithms

[R614] D’Agostino, R. B. (1971), “An omnibus test of normality for moderate and large sample size”, Biometrika, 58, 341-348

[R615] D’Agostino, R. and Pearson, E. S. (1973), “Tests for departure from normality”, Biometrika, 60, 613-622

[R635] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000. Section 2.2.24.1

[R637] R. B. D'Agostino, A. J. Belanger and R. B. D'Agostino Jr., "A suggestion for using powerful and informative tests of normality", American Statistician 44, pp. 316-321, 1990.

[R647] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000.

[R616] S. E. Maxwell and H. D. Delaney, "Designing Experiments and Analyzing Data: A Model Comparison Perspective", Wadsworth, 1990.

[R591] "Interquartile range" https://en.wikipedia.org/wiki/Interquartile_range

[R592] "Robust measures of scale" https://en.wikipedia.org/wiki/Robust_measures_of_scale

[R593] "Quantile" https://en.wikipedia.org/wiki/Quantile

[R574] Lowry, Richard. "Concepts and Applications of Inferential Statistics". Chapter 14. http://faculty.vassar.edu/lowry/ch14pt1.html

[R575] Heiman, G.W. Research Methods in Statistics. 2002.

[R576] McDonald, G. H. "Handbook of Biological Statistics", One-way ANOVA. http://www.biostathandbook.com/onewayanova.html

[R638] Zwillinger, D. and Kokoska, S. (2000). CRC Standard Probability and Statistics Tables and Formulae. Chapman & Hall: New York. 2000. Section 14.7

[R618] J. Lev, "The Point Biserial Coefficient of Correlation", Ann. Math. Statist., Vol. 20, no.1, pp. 125-126, 1949.

[R619] R.F. Tate, "Correlation Between a Discrete and a Continuous Variable. Point-Biserial Correlation.", Ann. Math. Statist., Vol. 25, np. 3, pp. 603-607, 1954.

[R620] http://onlinelibrary.wiley.com/doi/10.1002/9781118445112.stat06227/full

[R595] Maurice G. Kendall, "A New Measure of Rank Correlation", Biometrika Vol. 30, No. 1/2, pp. 81-93, 1938.

[R596] Maurice G. Kendall, "The treatment of ties in ranking problems", Biometrika Vol. 33, No. 3, pp. 239-251. 1945.

[R597] Gottfried E. Noether, "Elements of Nonparametric Statistics", John Wiley & Sons, 1967.

[R598] Peter M. Fenwick, "A new data structure for cumulative frequency tables", Software: Practice and Experience, Vol. 24, No. 3, pp. 327-336, 1994.

[R648] Sebastiano Vigna, "A weighted correlation index for rankings with ties", Proceedings of the 24th international conference on World Wide Web, pp. 1166-1176, ACM, 2015.

[R649] W.R. Knight, "A Computer Method for Calculating Kendall's Tau with Ungrouped Data", Journal of the American Statistical Association, Vol. 61, No. 314, Part 1, pp. 436-439, 1966.

[R650] Grace S. Shieh. "A weighted Kendall's tau statistic", Statistics & Probability Letters, Vol. 39, No. 1, pp. 17-24, 1998.

[R639] P.K. Sen, "Estimates of the regression coefficient based on Kendall's tau", J. Am. Stat. Assoc., Vol. 63, pp. 1379-1389, 1968.

[R640] H. Theil, "A rank-invariant method of linear and polynomial regression analysis I, II and III", Nederl. Akad. Wetensch., Proc. 53:, pp. 386-392, pp. 521-525, pp. 1397-1412, 1950.

[R641] W.L. Conover, "Practical nonparametric statistics", 2nd ed., John Wiley and Sons, New York, pp. 493.

[R643] http://en.wikipedia.org/wiki/T-test#Independent_two-sample_t-test

[R644] http://en.wikipedia.org/wiki/Welch%27s_t_test

[R645] http://en.wikipedia.org/wiki/T-test#Independent_two-sample_t-test

[R646] http://en.wikipedia.org/wiki/Welch%27s_t_test

[R567] Lowry, Richard. "Concepts and Applications of Inferential Statistics". Chapter 8. http://faculty.vassar.edu/lowry/ch8pt1.html

[R568] "Chi-squared test", http://en.wikipedia.org/wiki/Chi-squared_test

[R621] Lowry, Richard. "Concepts and Applications of Inferential Statistics". Chapter 8. http://faculty.vassar.edu/lowry/ch8pt1.html

[R622] "Chi-squared test", http://en.wikipedia.org/wiki/Chi-squared_test

[R623] "G-test", http://en.wikipedia.org/wiki/G-test

[R624] Sokal, R. R. and Rohlf, F. J. "Biometry: the principles and practice of statistics in biological research", New York: Freeman (1981)

[R625] Cressie, N. and Read, T. R. C., "Multinomial Goodness-of-Fit Tests", J. Royal Stat. Soc. Series B, Vol. 46, No. 3 (1984), pp. 440-464.

[R642] Siegel, S. (1956) Nonparametric Statistics for the Behavioral Sciences. New York: McGraw-Hill.

[R629] "Ranking", http://en.wikipedia.org/wiki/Ranking

[R630] http://en.wikipedia.org/wiki/Wilcoxon_rank-sum_test

[R651] http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[R599] W. H. Kruskal & W. W. Wallis, "Use of Ranks in One-Criterion Variance Analysis", Journal of the American Statistical Association, Vol. 47, Issue 260, pp. 583-621, 1952.

[R600] http://en.wikipedia.org/wiki/Kruskal-Wallis_one-way_analysis_of_variance

[R582] http://en.wikipedia.org/wiki/Friedman_test

[R569] https://en.wikipedia.org/wiki/Fisher%27s_method

[R570] http://en.wikipedia.org/wiki/Fisher's_method#Relation_to_Stouffer.27s_Z-score_method

[R571] Whitlock, M. C. "Combining probability from independent tests: the weighted Z-method is superior to Fisher's approach." Journal of Evolutionary Biology 18, no. 5 (2005): 1368-1373.

[R572] Zaykin, Dmitri V. "Optimally weighted Z-test is a powerful method for combining probabilities in meta-analysis." Journal of Evolutionary Biology 24, no. 8 (2011): 1836-1841.

[R573] https://en.wikipedia.org/wiki/Extensions_of_Fisher%27s_method

[R594] Jarque, C. and Bera, A. (1980) "Efficient tests for normality, homoscedasticity and serial independence of regression residuals", 6 Econometric Letters 255-259.

[R554] Sprent, Peter and N.C. Smeeton. Applied nonparametric statistical methods. 3rd ed. Chapman and Hall/CRC. 2001. Section 5.8.2.

[R556] http://www.itl.nist.gov/div898/handbook/eda/section3/eda357.htm

[R557] Snedecor, George W. and Cochran, William G. (1989), Statistical Methods, Eighth Edition, Iowa State University Press.

[R558] Park, C. and Lindsay, B. G. (1999). Robust Scale Estimation and Hypothesis Testing based on Quadratic Inference Function. Technical Report #99-03, Center for Likelihood Studies, Pennsylvania State University.

[R559] Bartlett, M. S. (1937). Properties of Sufficiency and Statistical Tests. Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 160, No.901, pp. 268-282.

[R603] http://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm

[R604] Levene, H. (1960). In Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling, I. Olkin et al. eds., Stanford University Press, pp. 278-292.

[R605] Brown, M. B. and Forsythe, A. B. (1974), Journal of the American Statistical Association, 69, 364-367

[R631] http://www.itl.nist.gov/div898/handbook/prc/section2/prc213.htm

[R632] Shapiro, S. S. & Wilk, M.B (1965). An analysis of variance test for normality (complete samples), Biometrika, Vol. 52, pp. 591-611.

[R633] Razali, N. M. & Wah, Y. B. (2011) Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests, Journal of Statistical Modeling and Analytics, Vol. 2, pp. 21-33.

[R634] ALGORITHM AS R94 APPL. STATIST. (1995) VOL. 44, NO. 4.

[R547] http://www.itl.nist.gov/div898/handbook/prc/section2/prc213.htm

[R548] Stephens, M. A. (1974). EDF Statistics for Goodness of Fit and Some Comparisons, Journal of the American Statistical Association, Vol. 69, pp. 730-737.

[R549] Stephens, M. A. (1976). Asymptotic Results for Goodness-of-Fit Statistics with Unknown Parameters, Annals of Statistics, Vol. 4, pp. 357-369.

[R550] Stephens, M. A. (1977). Goodness of Fit for the Extreme Value Distribution, Biometrika, Vol. 64, pp. 583-588.

[R551] Stephens, M. A. (1977). Goodness of Fit with Special Reference to Tests for Exponentiality , Technical Report No. 262, Department of Statistics, Stanford University, Stanford, CA.

[R552] Stephens, M. A. (1979). Tests of Fit for the Logistic Distribution Based on the Empirical Distribution Function, Biometrika, Vol. 66, pp. 591-595.

[R553] Scholz, F. W and Stephens, M. A. (1987), K-Sample Anderson-Darling Tests, Journal of the American Statistical Association, Vol. 82, pp. 918-924.

[R560] http://en.wikipedia.org/wiki/Binomial_test

[R578] http://www.stat.psu.edu/~bgl/center/tr/TR993.ps

[R579] Fligner, M.A. and Killeen, T.J. (1976). Distribution-free two-sample tests for scale. 'Journal of the American Statistical Association.' 71(353), 210-213.

[R580] Park, C. and Lindsay, B. G. (1999). Robust Scale Estimation and Hypothesis Testing based on Quadratic Inference Function. Technical Report #99-03, Center for Likelihood Studies, Pennsylvania State University.

[R581] Conover, W. J., Johnson, M. E. and Johnson M. M. (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf biding data. Technometrics, 23(4), 351-361.

[R607] Mood, A. M., Introduction to the Theory of Statistics. McGraw-Hill (1950), pp. 394-399.

[R608] Zar, J. H., Biostatistical Analysis, 5th ed. Prentice Hall (2010). See Sections 8.12 and 10.15.

[R564] "Contingency table", http://en.wikipedia.org/wiki/Contingency_table

[R565] "Pearson's chi-squared test", http://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test

[R566] Cressie, N. and Read, T. R. C., "Multinomial Goodness-of-Fit Tests", J. Royal Stat. Soc. Series B, Vol. 46, No. 3 (1984), pp. 440-464.

[R626] J.J. Filliben, "The Probability Plot Correlation Coefficient Test for Normality", Technometrics, Vol. 17, pp. 111-117, 1975.

[R627] http://www.itl.nist.gov/div898/handbook/eda/section3/ppccplot.htm

[R610] Lowry, Richard. "Concepts and Applications of Inferential Statistics". Chapter 8. http://faculty.vassar.edu/lowry/ch8pt1.html

[R611] "Chi-squared test", http://en.wikipedia.org/wiki/Chi-squared_test

[R612] *R* statistical software: http://www.r-project.org/

[R613] *R* `quantile` function: http://stat.ethz.ch/R-manual/R-devel/library/stats/html/quantile.html

[R583] D.W. Scott, “Multivariate Density Estimation: Theory, Practice, and Visualization”, John Wiley & Sons, New York, Chicester, 1992.

[R584] B.W. Silverman, “Density Estimation for Statistics and Data Analysis”, Vol. 26, Monographs on Statistics and Applied Probability, Chapman and Hall, London, 1986.

[R585] B.A. Turlach, “Bandwidth Selection in Kernel Density Estimation: A Review”, CORE and Institut de Statistique, Vol. 19, pp. 1-33, 1993.

[R586] D.M. Bashtannyk and R.J. Hyndman, “Bandwidth selection for kernel conditional density estimation”, Computational Statistics & Data Analysis, Vol. 36, pp. 279-298, 2001.