Example Programs for CVODE v3.2.1

Alan C. Hindmarsh and Radu Serban
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

October 11, 2018

UCRL-SM-208110

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Approved for public release; further dissemination unlimited

Contents

1 Introduction

This report is intended to serve as a companion document to the User Documentation of CVODE [2]. It provides details, with listings, on the example programs supplied with the CVODE distribution package.

The CVODE distribution contains examples of six types: serial C examples, parallel C examples, serial and parallel FORTRAN examples, an OpenMP example, and a *hypre* example. With the exception of "demo"-type example files, the names of all the examples distributed with SUNDIALS are of the form `[slv][PbName]_[ls]_[prec]_[p]`, where

[slv] identifies the solver (for CVODE examples this is `cv`, while for FCVODE examples, this is `fcv`);

[PbName] identifies the problem;

[ls] identifies the linear solver module used (for examples using functional iteration for the nonlinear system solver, `non` specifies that no linear solver was used);

[prec] indicates the CVODE preconditioner module used, `bp` for CVBANDPRE or `bbd` for CVBBDPRE (only if applicable, for examples using a Krylov linear solver);

[p] indicates an example using the parallel vector module NVECTOR_PARALLEL.

The following lists summarize all examples distributed with CVODE.

Supplied in the *srcdir*`/examples/cvode/serial` directory are the following serial examples (using the NVECTOR_SERIAL module):

- `cvRoberts_dns` solves a chemical kinetics problem consisting of three rate equations. This program solves the problem with the BDF method and Newton iteration, with the SUNLINSOL_DENSE linear solver, CVDLS interface, and a user-supplied Jacobian routine. It also uses the rootfinding feature of CVODE.
- `cvRoberts_dns_constraints` is the same as `cvRoberts_dns` but imposes the constraint $u \geq 0.0$ for all components.
- `cvRoberts_dnsL` is the same as `cvRoberts_dns` but uses the Lapack implementation of SUNLINSOL_LAPACKDENSE.
- `cvRoberts_dns_uw` is the same as `cvRoberts_dns` but demonstrates the user-supplied error weight function feature of CVODE.
- `cvRoberts_klu` is the same as `cvRoberts_dns` but uses the KLU sparse direct linear solver, SUNLINSOL_KLU.
- `cvRoberts_sps` is the same as `cvRoberts_dns` but uses the SUPERLUMT sparse direct linear solver, SUNLINSOL_SUPERLUMT (with one thread).
- `cvAdvDiff_bnd` solves the semi-discrete form of an advection-diffusion equation in 2-D. This program solves the problem with the BDF method and Newton iteration, with the SUNLINSOL_BAND linear solver, CVDLS interface, and a user-supplied Jacobian routine.

- `cvAdvDiff_bndL` is the same as `cvAdvDiff_bnd` but uses the Lapack implementation of SUNLINSOL_LAPACKBAND.

- `cvDiurnal_kry` solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D.
  The problem is solved with the BDF/GMRES method (i.e. using the SUNLINSOL_SPGMR linear solver and CVSPILS interface) and the block-diagonal part of the Newton matrix as a left preconditioner. A copy of the block-diagonal part of the Jacobian is saved and conditionally reused within the preconditioner setup routine.

- `cvDiurnal_kry_bp` solves the same problem as `cvDiurnal_kry`, with the BDF/GMRES method and a banded preconditioner, generated by difference quotients, using the module CVBANDPRE.
  The problem is solved twice: with preconditioning on the left, then on the right.

- `cvDirectDemo_ls` is a demonstration program for CVODE with direct linear solvers.
  Two separate problems are solved using both the Adams and BDF linear multistep methods in combination with functional and Newton iterations.
  The first problem is the Van der Pol oscillator for which the Newton iteration cases use the following types of Jacobian approximations: (1) dense, user-supplied, (2) dense, difference-quotient approximation, (3) diagonal, with difference-quotient approximation. The second problem is a linear ODE with a banded lower triangular matrix derived from a 2-D advection PDE. In this case, the Newton iteration cases use the following types of Jacobian approximation: (1) banded, user-supplied, (2) banded, difference-quotient approximation, (3) diagonal, difference-quotient approximation.

- `cvKrylovDemo_ls` solves the same problem as `cvDiurnal_kry`, with the BDF method, but with three Krylov linear solvers: SUNLINSOL_SPGMR, SUNLINSOL_SPBCGS, and SUNLINSOL_SPTFQMR.

- `cvKrylovDemo_prec` is a demonstration program with the GMRES linear solver.
  This program solves a stiff ODE system that arises from a system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions.
  The ODE system is solved using Newton iteration, the SUNLINSOL_SPGMR linear solver (scaled preconditioned GMRES), and CVSPILS interface.
  The preconditioner matrix used is the product of two matrices: (1) a matrix, only defined implicitly, based on a fixed number of Gauss-Seidel iterations using the diffusion terms only; and (2) a block-diagonal matrix based on the partial derivatives of the interaction terms only, using block-grouping.
  Four different runs are made for this problem. The product preconditoner is applied on the left and on the right. In each case, both the modified and classical Gram-Schmidt options are tested.

- `cvHeat2D_klu` solves a discretized 2D heat equation using the KLU sparse-direct linear solver, SUNLINSOL_KLU.

Supplied in the *srcdir*`/examples/cvode/parallel` directory are the following four parallel examples (using the NVECTOR_PARALLEL module):

- `cvAdvDiff_non_p` solves the semi-discrete form of a 1-D advection-diffusion equation. This program solves the problem with the option for nonstiff systems, i.e. Adams method and functional iteration.
- `cvAdvDiff_diag_p` solves the same problem as `cvAdvDiff_non_p`, with the Adams method, but with Newton iteration and the `CVDiag` linear solver.
- `cvDiurnal_kry_p` is a parallel implementation of `cvDiurnal_kry`.
- `cvDiurnal_kry_bbd_p` solves the same problem as `cvDiurnal_kry_p`, with BDF and the GMRES linear solver, using a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module CVBBDPRE.

Supplied in the *srcdir*`/examples/cvode/C_openmp` directory is `cvAdvDiff_bnd_omp`, an example which solves the same problem as `cvAdvDiff_bnd` but with the OpenMP NVECTOR module.

Supplied in the *srcdir*`/examples/cvode/parhyp` directory is an example `cvAdvDiff_non_ph`, which solves the same problem as `cvAdvDiff_non_p` but with *hypre* vectors instead of SUNDIALS parallel vectors.

As part of the FCVODE module, in the directories *srcdir*`/examples/cvode/fcmix_serial` and *srcdir*`/examples/cvode/fcmix_parallel`, are the following examples for the FORTRAN-C interface. The first five of these are serial, while the last three are parallel.

- `fcvRoberts_dns` is a serial chemical kinetics example (`BDF`/SUNLINSOL_DENSE) with rootfinding.
- `fcvRoberts_dns_constraints` is the same as `fcvRoberts_dns` but but imposes the constraint $u \geq 0.0$ for all components.
- `fcvRoberts_dnsL` is the same as `fcvRoberts_dns` but uses the Lapack implementation of SUNLINSOL_LAPACKDENSE.
- `fcvAdvDiff_bnd` is a serial advection-diffusion example (`BDF`/SUNLINSOL_BAND).
- `fcvDiurnal_kry` is a serial kinetics-transport example (`BDF`/SUNLINSOL_SPGMR).
- `fcvDiurnal_kry_bp` is the `fcvDiurnal_kry` example with FCVBP.
- `fcvDiag_non_p` is a nonstiff parallel diagonal ODE example (`ADAMS`/`FUNCTIONAL`).
- `fcvDiag_kry_p` is a stiff parallel diagonal ODE example (`BDF`/SUNLINSOL_SPGMR).
- `fcvDiag_kry_bbd_p` is the same as the `fcvDiag_kry_p` example but using the FCVBBD module.

In the following sections, we give detailed descriptions of some (but not all) of these examples. We also give our output files for each of these examples, but users should be cautioned that their results may differ slightly from these. Differences in solution values may differ within

the tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

The final section of this report describes a set of tests done with the parallel version of CVODE, using a problem based on the `cvDiurnal_kry`/`cvDiurnal_kry_p` example.

In the descriptions below, we make frequent references to the CVODE User Document [2]. All citations to specific sections (e.g. §4.2) are references to parts of that User Document, unless explicitly stated otherwise.

Note. The examples in the CVODE distribution are written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see Appendix A in the User Guide). As a consequence, they contain portions of code that will not be typically present in a user program. For example, all C example programs make use of the variables `SUNDIALS_EXTENDED_PRECISION` and `SUNDIALS_DOUBLE_PRECISION` to test if the solver libraries were built in extended or double precision, and use the appropriate conversion specifiers in `printf` functions.

2 Serial example problems

2.1 A dense example: cvRoberts_dns

As an initial illustration of the use of the CVODE package for the integration of IVP ODEs, we give a sample program called `cvRoberts_dns.c`. It uses the CVODE direct linear solver interface CVDLS with dense matrix and linear solver modules (SUNMATRIX_DENSE and SUNLINSOL_DENSE) and the NVECTOR_SERIAL module (which provides a serial implementation of NVECTOR) in the solution of a 3-species chemical kinetics problem.

The problem consists of the following three rate equations:

$$\begin{aligned} \dot{y}_1 &= -0.04 \cdot y_1 + 10^4 \cdot y_2 \cdot y_3 \\ \dot{y}_2 &= 0.04 \cdot y_1 - 10^4 \cdot y_2 \cdot y_3 - 3 \cdot 10^7 \cdot y_2^2 \\ \dot{y}_3 &= 3 \cdot 10^7 \cdot y_2^2 \end{aligned} \tag{1}$$

on the interval $t \in [0,\ 4 \cdot 10^{10}]$, with initial conditions $y_1(0) = 1.0$, $y_2(0) = y_3(0) = 0.0$. While integrating the system, we also use the rootfinding feature to find the points at which $y_1 = 10^{-4}$ or at which $y_3 = 0.01$.

For the source we give a rather detailed explanation of the parts of the program and their interaction with CVODE.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in CVODE header files. The `sundials_types.h` file provides the definition of the type `realtype` (see §4.2 for details). For now, it suffices to read `realtype` as `double`. The `cvode.h` file provides prototypes for the CVODE functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `CVode`. The `cvode_direct.h` file provides the prototype for the `CVDlsAttachLinearSolver` function. The `sunlinsol_dense.h` file is the header file for the dense implementation of the SUNLINSOL module and includes definitions of the `SUNLinearSolver` type. Similarly, the `sunmatrix_dense.h` file is the header file for the dense implementation of the SUNMATRIX module, including definitions of the `SUNMatrix` type as well as macros and functions to access matrix components. We have explicitly included `sunmatrix_dense.h`, but this is not necessary because it is included by `sunlinsol_dense.h`. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions.

This program includes two user-defined accessor macros, `Ith` and `IJth`, that are useful in writing the problem functions in a form closely matching the mathematical description of the ODE system, i.e. with components numbered from 1 instead of from 0. The `Ith` macro is used to access components of a vector of type `N_Vector` with a serial implementation. It is defined using the NVECTOR_SERIAL accessor macro `NV_Ith_S` which numbers components starting with 0. The `IJth` macro is used to access elements of a dense matrix of type `SUNMatrix`. It is similarly defined using the SUNMATRIX_DENSE accessor macro `SM_ELEMENT_D` which numbers matrix rows and columns starting with 0. The macro `NV_Ith_S` is fully described in §6.1. The macro `SM_ELEMENT_D` is fully described in §**??**.

Next, the program includes some problem-specific constants, which are isolated to this early location to make it easy to change them as needed. The program prologue ends with prototypes of four private helper functions and the three user-supplied functions that are called by CVODE.

The main program begins with some dimensions and type declarations, including use of the generic types N_Vector, SUNMatrix and SUNLinearSolver. The next several lines allocate memory for the y and abstol vectors using N_VNew_Serial with a length argument of NEQ (= 3). The lines following that load the initial values of the dependendent variable vector into y and the absolute tolerances into abstol using the Ith macro.

The calls to N_VNew_Serial, and also later calls to CVode*** functions, make use of a private function, check_flag, which examines the return value and prints a message if there was a failure. The check_flag function was written to be used for any serial SUNDIALS application.

The call to CVodeCreate creates the CVODE solver memory block, specifying the CV_BDF integration method with CV_NEWTON iteration. Its return value is a pointer to that memory block for this problem. In the case of failure, the return value is NULL. This pointer must be passed in the remaining calls to CVODE functions.

The call to CVodeInit allocates and initializes the solver memory block. Its arguments include the name of the C function f defining the right-hand side function $f(t, y)$, and the initial values of t and y. The call to CVodeSVtolerances specifies a vector of absolute tolerances, and includes the value of the relative tolerance reltol and the absolute tolerance vector abstol. See §4.5.1 and §4.5.2 for full details of these calls.

The call to CVodeRootInit specifies that a rootfinding problem is to be solved along with the integration of the ODE system, that the root functions are specified in the function g, and that there are two such functions. Specifically, they are set to $y_1 - 0.0001$ and $y_3 - 0.01$, respectively. See §4.5.4 for a detailed description of this call.

The call to SUNDenseMatrix (see §??) creates a NEQ×NEQ dense SUNMATRIX object to use within the Newton solve in CVODE. The following call to SUNDenseLinearSolver (see §??) creates the dense SUNLINSOL object that will perform the linear solves within the Newton method. These are attached to the CVDLS direct linear solver interface with the call to CVDlsSetLinearSolver (see §4.5.3), and the subsequent call to CVDlsSetJacFn (see §4.5.6) specifies the analytic Jacobian supplied by the user-supplied function Jac.

The actual solution of the ODE initial value problem is accomplished in the loop over values of the output time tout. In each pass of the loop, the program calls CVode in the CV_NORMAL mode, meaning that the integrator is to take steps until it overshoots tout and then interpolate to t =tout, putting the computed value of y(tout) into y, with t = tout. The return value in this case is CV_SUCCESS. However, if CVode finds a root before reaching the next value of tout, it returns CV_ROOT_RETURN and stores the root location in t and the solution there in y. In either case, the program prints t and y. In the case of a root, it calls CVodeGetRootInfo to get a length-2 array rootsfound of bits showing which root function was found to have a root. If CVode returned any negative value (indicating a failure), the program breaks out of the loop. In the case of a CV_SUCCESS return, the value of tout is advanced (multiplied by 10) and a counter (iout) is advanced, so that the loop can be ended when that counter reaches the preset number of output times, NOUT = 12. See §4.5.5 for full details of the call to CVode.

Finally, the main program calls PrintFinalStats to get and print all of the relevant statistical quantities. It then calls NV_Destroy to free the vectors y and abstol, CVodeFree to free the CVODE memory block, SUNLinSolFree to free the linear solver memory, and SUNMatDestroy to free the matrix A.

The function PrintFinalStats used here is actually suitable for general use in applications of CVODE to any problem with a direct linear solver. It calls various CVodeGet*** and CVDlsGet*** functions to obtain the relevant counters, and then prints them. Specif-

ically, these are: the cumulative number of steps (`nst`), the number of `f` evaluations (`nfe`) (excluding those for difference-quotient Jacobian evaluations), the number of matrix factorizations (`nsetups`), the number of `f` evaluations for Jacobian evaluations (`nfeLS` = 0 here), the number of Jacobian evaluations (`nje`), the number of nonlinear (Newton) iterations (`nni`), the number of nonlinear convergence failures (`ncfn`), the number of local error test failures (`netf`), and the number of `g` (root function) evaluations (`nge`). These optional outputs are described in §4.5.8.

The function `f` is a straightforward expression of the ODEs. It uses the user-defined macro `Ith` to extract the components of `y` and to load the components of `ydot`. See §4.6.1 for a detailed specification of `f`.

Similarly, the function `g` defines the two functions, g_0 and g_1, whose roots are to be found. See §4.6.4 for a detailed description of the `g` function.

The function `Jac` sets the nonzero elements of the Jacobian as a dense matrix. (Zero elements need not be set because `J` is preset to zero.) It uses the user-defined macro `IJth` to reference the elements of a dense matrix of type SUNMATRIX. Here the problem size is small, so we need not worry about the inefficiency of using `NV_Ith_S` and `SM_ELEMENT_D` to access `N_Vector` and SUNMATRIX_DENSE elements. Note that in this example, `Jac` only accesses the `y` and `J` arguments. See §4.6.5 for a detailed description of the `Jac` function.

The output generated by `cvRoberts_dns` is shown below. It shows the output values at the 12 preset values of `tout`. It also shows the two root locations found, first at a root of g_1, and then at a root of g_0.

cvRoberts_dns sample output

```
3-species kinetics problem

At t = 2.6391e-01      y =  9.899653e-01    3.470564e-05    1.000000e-02
    rootsfound[] =   0   1
At t = 4.0000e-01      y =  9.851641e-01    3.386242e-05    1.480205e-02
At t = 4.0000e+00      y =  9.055097e-01    2.240338e-05    9.446793e-02
At t = 4.0000e+01      y =  7.158017e-01    9.185037e-06    2.841892e-01
At t = 4.0000e+02      y =  4.505360e-01    3.223271e-06    5.494608e-01
At t = 4.0000e+03      y =  1.832299e-01    8.944378e-07    8.167692e-01
At t = 4.0000e+04      y =  3.898902e-02    1.622006e-07    9.610108e-01
At t = 4.0000e+05      y =  4.936383e-03    1.984224e-08    9.950636e-01
At t = 4.0000e+06      y =  5.168093e-04    2.068293e-09    9.994832e-01
At t = 2.0790e+07      y =  1.000000e-04    4.000397e-10    9.999000e-01
    rootsfound[] =  -1   0
At t = 4.0000e+07      y =  5.202440e-05    2.081083e-10    9.999480e-01
At t = 4.0000e+08      y =  5.201061e-06    2.080435e-11    9.999948e-01
At t = 4.0000e+09      y =  5.258603e-07    2.103442e-12    9.999995e-01
At t = 4.0000e+10      y =  6.934511e-08    2.773804e-13    9.999999e-01

Final Statistics:
nst = 542    nfe  = 755    nsetups = 107    nfeLS = 0      nje = 11
nni = 751    ncfn = 0      netf = 22     nge = 570
```

2.2 A banded example: cvAdvDiff_bnd

The example program `cvAdvDiff_bnd.c` solves the semi-discretized form of the 2-D advection-diffusion equation

$$\partial v/\partial t = \partial^2 v/\partial x^2 + .5\partial v/\partial x + \partial^2 v/\partial y^2 \tag{2}$$

on a rectangle, with zero Dirichlet boundary conditions. The PDE is discretized with standard central finite differences on a (MX+2) × (MY+2) mesh, giving an ODE system of size MX*MY. The discrete value v_{ij} approximates v at $x = i\Delta x$, $y = j\Delta y$. The ODEs are

$$\frac{dv_{ij}}{dt} = f_{ij} = \frac{v_{i-1,j} - 2v_{ij} + v_{i+1,j}}{(\Delta x)^2} + .5\frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} + \frac{v_{i,j-1} - 2v_{ij} + v_{i,j+1}}{(\Delta y)^2}, \qquad (3)$$

where $1 \le i \le$MX and $1 \le j \le$MY. The boundary conditions are imposed by taking $v_{ij} = 0$ above if $i = 0$ or MX+1, or if $j = 0$ or MY+1. If we set $u_{(j-1)+(i-1)*\texttt{MY}} = v_{ij}$, so that the ODE system is $\dot{u} = f(u)$, then the system Jacobian $J = \partial f/\partial u$ is a band matrix with upper and lower half-bandwidths both equal to MY. In the example, we take MX = 10 and MY = 5.

The cvAdvDiff_bnd.c program includes files sunmatrix_band.h, sunlinsol_band.h and cvode_direct.h in order to use the SUNLINSOL_BAND linear solver. The sunmatrix_band.h file contains the definition of the banded SUNMATRIX type, and the SM_COLUMN_B and SM_COLUMN_ELEMENT_B macros for accessing banded matrix elements (see §??). The sunlinsol_band.h file contains the definition of the banded SUNLINSOL type. We note that have explicitly included sunmatrix_band.h, but this is not necessary because it is included by sunlinsol_band.h. The cvode_direct.h file contains the prototype for the CVDlsSetLinearSolver routine. The file nvector_serial.h is included for the definition of the serial N_Vector type.

The include lines at the top of the file are followed by definitions of problem constants which include the x and y mesh dimensions, MX and MY, the number of equations NEQ, the scalar absolute tolerance ATOL, the initial time T0, and the initial output time T1.

Spatial discretization of the PDE naturally produces an ODE system in which equations are numbered by mesh coordinates (i, j). The user-defined macro IJth isolates the translation for the mathematical two-dimensional index to the one-dimensional N_Vector index and allows the user to write clean, readable code to access components of the dependent variable. The NV_DATA_S macro returns the component array for a given N_Vector, and this array is passed to IJth in order to do the actual N_Vector access.

The type UserData is a pointer to a structure containing problem data used in the f and Jac functions. This structure is allocated and initialized at the beginning of main. The pointer to it, called data, is passed to CVodeSetUserData, and as a result it will be passed back to the f and Jac functions each time they are called. The use of the data pointer eliminates the need for global program data.

The main program is straightforward. The CVodeCreate call specifies the CV_BDF method with a CV_NEWTON iteration. Following the CVodeInit call, the call to CVodeSStolerances indicates scalar relative and absolute tolerances, and values reltol and abstol are passed. The call to SUNBandMatrix (see §??) creates a banded SUNMATRIX Jacobian template, and specifies that both half-bandwidths of the Jacobian are equal to MY. The calls to SUNBandLinearSolver (see §??) and CVDlsSetLinearSolver (see §4.5.3) specifies the SUNLINSOL_BAND linear solver to the CVDLS interface. The call to CVDlsSetJacFn (see §4.5.6) specifies that a user-supplied Jacobian function Jac is to be used.

The actual solution of the problem is performed by the call to CVode within the loop over the output times tout. The max-norm of the solution vector (from a call to N_VMaxNorm) and the cumulative number of time steps (from a call to CVodeGetNumSteps) are printed at each output time. Finally, the calls to PrintFinalStats, N_VDestroy, and CVodeFree print statistics and free problem memory.

Following the main program in the cvAdvDiff_bnd.c file are definitions of five functions: f, Jac, SetIC, PrintHeader, PrintOutput, PrintFinalStats, and check_flag. The last five functions are called only from within the cvAdvDiff_bnd.c file. The SetIC function sets

the initial dependent variable vector; `PrintHeader` prints the heading of the output page; `PrintOutput` prints a line of solution output; `PrintFinalStats` gets and prints statistics at the end of the run; and `check_flag` aids in checking return values. The statistics printed include counters such as the total number of steps (`nst`), `f` evaluations (excluding those for Jaobian evaluations) (`nfe`), LU decompositions (`nsetups`), `f` evaluations for difference-quotient Jacobians (`nfeLS` = 0 here), Jacobian evaluations (`nje`), and nonlinear iterations (`nni`). These optional outputs are described in §4.5.8. Note that `PrintFinalStats` is suitable for general use in applications of CVODE to any problem with a direct linear solver.

The `f` function implements the central difference approximation (3) with u identically zero on the boundary. The constant coefficients $(\Delta x)^{-2}$, $.5(2\Delta x)^{-1}$, and $(\Delta y)^{-2}$ are computed only once at the beginning of `main`, and stored in the locations `data->hdcoef`, `data->hacoef`, and `data->vdcoef`, respectively. When `f` receives the `data` pointer (renamed `user_data` here), it pulls out these values from storage in the local variables `hordc`, `horac`, and `verdc`. It then uses these to construct the diffusion and advection terms, which are combined to form `udot`. Note the extra lines setting out-of-bounds values of u to zero.

The `Jac` function is an expression of the derivatives

$$\begin{aligned} \partial f_{ij}/\partial v_{ij} &= -2[(\Delta x)^{-2} + (\Delta y)^{-2}] \\ \partial f_{ij}/\partial v_{i\pm1,j} &= (\Delta x)^{-2} \pm .5(2\Delta x)^{-1}, \quad \partial f_{ij}/\partial v_{i,j\pm1} = (\Delta y)^{-2} \ . \end{aligned}$$

This function loads the Jacobian by columns, and like `f` it makes use of the preset coefficients in `data`. It loops over the mesh points (`i`,`j`). For each such mesh point, the one-dimensional index `k = j-1 + (i-1)*MY` is computed and the `k`th column of the Jacobian matrix J is set. The row index k' of each component $f_{i',j'}$ that depends on $v_{i,j}$ must be identified in order to load the corresponding element. The elements are loaded with the `SM_COLUMN_ELEMENT_B` macro. Note that the formula for the global index k implies that decreasing (increasing) `i` by 1 corresponds to decreasing (increasing) `k` by `MY`, while decreasing (increasing) `j` by 1 corresponds of decreasing (increasing) `k` by 1. These statements are reflected in the arguments to `SM_COLUMN_ELEMENT_B`. The first argument passed to the `SM_COLUMN_ELEMENT_B` macro is a pointer to the diagonal element in the column to be accessed. This pointer is obtained via a call to the `SM_COLUMN_B` macro and is stored in `kthCol` in the `Jac` function. When setting the components of J we must be careful not to index out of bounds. The guards `(i != 1)` etc. in front of the calls to `SM_COLUMN_ELEMENT_B` prevent illegal indexing. See §4.6.5 for a detailed description of the `Jac` function.

The output generated by `cvAdvDiff_bnd` is shown below.

cvAdvDiff_bnd sample output

```
2-D Advection-Diffusion Equation
Mesh dimensions = 10 X 5
Total system size = 50
Tolerance parameters: reltol = 0   abstol = 1e-05

At t = 0      max.norm(u) =  8.954716e+01
At t = 0.10   max.norm(u) =  4.132889e+00   nst =   85
At t = 0.20   max.norm(u) =  1.039294e+00   nst =  103
At t = 0.30   max.norm(u) =  2.979829e-01   nst =  113
At t = 0.40   max.norm(u) =  8.765774e-02   nst =  120
At t = 0.50   max.norm(u) =  2.625637e-02   nst =  126
At t = 0.60   max.norm(u) =  7.830425e-03   nst =  130
At t = 0.70   max.norm(u) =  2.329387e-03   nst =  134
```

```
At t = 0.80    max.norm(u) =  6.953434e-04   nst =  137
At t = 0.90    max.norm(u) =  2.115983e-04   nst =  140
At t = 1.00    max.norm(u) =  6.556853e-05   nst =  142

Final Statistics:
nst = 142     nfe  = 174     nsetups = 23      nfeLS = 0       nje = 3
nni = 170     ncfn = 0       netf = 3
```

2.3 A Krylov example: cvDiurnal_kry

We give here an example that illustrates the use of CVODE with the Krylov method SPGMR, in the SUNLINSOL_SPGMR module, as the linear system solver through the CVSPILS interface.

This program solves the semi-discretized form of a pair of kinetics-advection-diffusion partial differential equations, which represent a simplified model for the transport, production, and loss of ozone and the oxygen singlet in the upper atmosphere. The problem includes nonlinear diurnal kinetics, horizontal advection and diffusion, and nonuniform vertical diffusion. The PDEs can be written as

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V\frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y}K_v(y)\frac{\partial c^i}{\partial y} + R^i(c^1, c^2, t) \quad (i = 1, 2) \ , \tag{4}$$

where the superscripts i are used to distinguish the two chemical species, and where the reaction terms are given by

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t)c^3 + q_4(t)c^2 \ , \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t)c^2 \ . \end{aligned} \tag{5}$$

The spatial domain is $0 \le x \le 20,\ 30 \le y \le 50$ (in km). The various constants and parameters are: $K_h = 4.0 \cdot 10^{-6},\ V = 10^{-3},\ K_v = 10^{-8}\exp(y/5),\ q_1 = 1.63 \cdot 10^{-16},\ q_2 = 4.66 \cdot 10^{-16},\ c^3 = 3.7 \cdot 10^{16}$, and the diurnal rate constants are defined as:

$$q_i(t) = \left\{ \begin{array}{ll} \exp[-a_i/\sin\omega t], & \text{for } \sin\omega t > 0 \\ 0, & \text{for } \sin\omega t \le 0 \end{array} \right\} \quad (i = 3, 4) \ ,$$

where $\omega = \pi/43200,\ a_3 = 22.62,\ a_4 = 7.601$. The time interval of integration is $[0, 86400]$, representing 24 hours measured in seconds.

Homogeneous Neumann boundary conditions are imposed on each boundary, and the initial conditions are

$$\begin{aligned} c^1(x, y, 0) = 10^6\alpha(x)\beta(y) \ , \quad c^2(x, y, 0) = 10^{12}\alpha(x)\beta(y) \ , \\ \alpha(x) = 1 - (0.1x - 1)^2 + (0.1x - 1)^4/2 \ , \\ \beta(y) = 1 - (0.1y - 4)^2 + (0.1y - 4)^4/2 \ . \end{aligned} \tag{6}$$

For this example, the equations (4) are discretized spatially with standard central finite differences on a 10×10 mesh, giving an ODE system of size 200.

Among the initial `#include` lines in this case are lines to include `sunlinsol_spgmr.h` and `sundials_math.h`. The first contains constants and function prototypes associated with the SUNLINSOL_SPGMR module, including the values of the `pretype` argument to `SUNSPGMR`. The inclusion of `sundials_math.h` is done to access the `SUNSQR` macro for the square of a `realtype` number.

The main program calls CVodeCreate specifying the CV_BDF method and CV_NEWTON iteration, and then calls CVodeInit, and CVodeSetSStolerances specifies the scalar tolerances. It calls SUNSPGMR to create the SPGMR linear solver with left preconditioning, and the default value (indicated by a zero argument) for maxl. It then calls CVSpilsSetLinearSolver (see §4.5.3) to attach this linear solver to the CVSPILS interface. The call to CVSpilsSetJacTimes specifies a user-supplied function for Jacobian-vector products (the NULL argument specifies that no Jacobian-vector setup routine is needed). Next, user-supplied preconditioner setup and solve functions, Precond and PSolve, are specified. See §4.5.6 for details on the CVSpilsSetPreconditioner function.

For a sequence of tout values, CVode is called in the CV_NORMAL mode, sampled output is printed, and the return value is tested for error conditions. After that, PrintFinalStats is called to get and print final statistics, and memory is freed by calls to N_VDestroy, FreeUserData, and CVodeFree. The printed statistics include various counters, such as the total numbers of steps (nst), of f evaluations (excluding those for Jv product evaluations) (nfe), of f evaluations for Jv evaluations (nfeLS), of nonlinear iterations (nni), of linear (Krylov) iterations (nli), of preconditioner setups (nsetups), of preconditioner evaluations (npe), and of preconditioner solves (nps), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator CVode, denoted lenrw and leniw, and those used by CVSPILS, denoted lenrwLS and leniwLS. All of these optional outputs are described in §4.5.8. The PrintFinalStats function is suitable for general use in applications of CVODE to any problem with an iterative linear solver.

Mathematically, the dependent variable has three dimensions: species number, x mesh point, and y mesh point. But in NVECTOR_SERIAL, a vector of type N_Vector works with a one-dimensional contiguous array of data components. The macro IJKth isolates the translation from three dimensions to one. Its use results in clearer code and makes it easy to change the underlying layout of the three-dimensional data. Here the problem size is 200, so we use the NV_DATA_S macro for efficient N_Vector access. The NV_DATA_S macro gives a pointer to the first component of an N_Vector which we pass to the IJKth macro to do an N_Vector access.

The preconditioner used here is the block-diagonal part of the true Newton matrix. It is generated and factored in the Precond routine (see §4.6.9) and backsolved in the PSolve routine (see §4.6.8). Its diagonal blocks are 2×2 matrices that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian itself, J_{bd}, is saved in separate storage each time it is generated, on calls to Precond with jok == SUNFALSE. On calls with jok == SUNTRUE, signifying that saved Jacobian data can be reused, the preconditioner $P = I - \gamma J_{bd}$ is formed from the saved matrix J_{bd} and factored. (A call to Precond with jok == SUNTRUE can only occur after a prior call with jok == SUNFALSE.) The Precond routine must also set the value of jcur, i.e. *jcurPtr, to SUNTRUE when J_{bd} is re-evaluated, and SUNFALSE otherwise, to inform CVSPILS of the status of Jacobian data.

We need to take a brief detour to explain one last important aspect of this program. While the generic SUNLINSOL_DENSE linear solver module serves as the interface to dense matrix solves for the main SUNDIALS solvers, the underlying algebraic operations operate on dense matrices with realtype ** as the underlying dense matrix type. To avoid the extra layer of function calls and dense matrix and linear solver data structures, cvDiurnal_kry.c uses underling small dense functions for all operations on the 2×2 preconditioner blocks. Thus it includes sundials_dense.h, and calls the small dense matrix functions newDenseMat, newIndexArray, denseCopy, denseScale, denseAddIdentity, denseGETRF, and denseGETRS.

The macro `IJth` defined near the top of the file is used to access individual elements in each preconditioner block, numbered from 1. The underlying dense algebra functions are available for CVODE user programs generally.

In addition to the functions called by CVODE, `cvDiurnal_kry.c` includes definitions of several private functions. These are: `AllocUserData` to allocate space for J_{bd}, P, and the pivot arrays; `InitUserData` to load problem constants in the `data` block; `FreeUserData` to free that block; `SetInitialProfiles` to load the initial values in `y`; `PrintOutput` to retreive and print selected solution values and statistics; `PrintFinalStats` to print statistics; and `check_flag` to check return values for error conditions.

The output generated by `cvDiurnal_kry.c` is shown below. Note that the number of preconditioner evaluations, `npe`, is much smaller than the number of preconditioner setups, `nsetups`, as a result of the Jacobian re-use scheme.

cvDiurnal_dns sample output

```
2-species diurnal advection-diffusion problem

t = 7.20e+03   no. steps = 219   order = 5   stepsize = 1.59e+02
c1 (bot.left/middle/top rt.) =    1.047e+04     2.964e+04     1.119e+04
c2 (bot.left/middle/top rt.) =    2.527e+11     7.154e+11     2.700e+11

t = 1.44e+04   no. steps = 251   order = 5   stepsize = 3.77e+02
c1 (bot.left/middle/top rt.) =    6.659e+06     5.316e+06     7.301e+06
c2 (bot.left/middle/top rt.) =    2.582e+11     2.057e+11     2.833e+11

t = 2.16e+04   no. steps = 277   order = 5   stepsize = 2.75e+02
c1 (bot.left/middle/top rt.) =    2.665e+07     1.036e+07     2.931e+07
c2 (bot.left/middle/top rt.) =    2.993e+11     1.028e+11     3.313e+11

t = 2.88e+04   no. steps = 307   order = 4   stepsize = 2.03e+02
c1 (bot.left/middle/top rt.) =    8.702e+06     1.292e+07     9.650e+06
c2 (bot.left/middle/top rt.) =    3.380e+11     5.029e+11     3.751e+11

t = 3.60e+04   no. steps = 338   order = 5   stepsize = 9.92e+01
c1 (bot.left/middle/top rt.) =    1.404e+04     2.029e+04     1.561e+04
c2 (bot.left/middle/top rt.) =    3.387e+11     4.894e+11     3.765e+11

t = 4.32e+04   no. steps = 393   order = 4   stepsize = 2.57e+02
c1 (bot.left/middle/top rt.) =   -7.891e-06    -1.123e-06    -8.723e-06
c2 (bot.left/middle/top rt.) =    3.382e+11     1.355e+11     3.804e+11

t = 5.04e+04   no. steps = 421   order = 5   stepsize = 4.96e+02
c1 (bot.left/middle/top rt.) =   -1.595e-08    -1.771e-06    -3.117e-08
c2 (bot.left/middle/top rt.) =    3.358e+11     4.930e+11     3.864e+11

t = 5.76e+04   no. steps = 439   order = 4   stepsize = 1.19e+02
c1 (bot.left/middle/top rt.) =   -2.127e-06    -1.513e-04    -2.951e-06
c2 (bot.left/middle/top rt.) =    3.320e+11     9.650e+11     3.909e+11

t = 6.48e+04   no. steps = 458   order = 5   stepsize = 6.60e+02
c1 (bot.left/middle/top rt.) =   -1.084e-09    -7.686e-08    -1.499e-09
c2 (bot.left/middle/top rt.) =    3.313e+11     8.922e+11     3.963e+11

t = 7.20e+04   no. steps = 469   order = 5   stepsize = 6.60e+02
c1 (bot.left/middle/top rt.) =   -1.093e-10    -7.736e-09    -1.510e-10
c2 (bot.left/middle/top rt.) =    3.330e+11     6.186e+11     4.039e+11
```

```
t = 7.92e+04   no. steps = 480   order = 5   stepsize = 6.60e+02
c1 (bot.left/middle/top rt.) =   -1.624e-12    -1.151e-10    -2.246e-12
c2 (bot.left/middle/top rt.) =    3.334e+11     6.669e+11     4.120e+11

t = 8.64e+04   no. steps = 491   order = 5   stepsize = 6.60e+02
c1 (bot.left/middle/top rt.) =    3.501e-14     2.474e-12     4.816e-14
c2 (bot.left/middle/top rt.) =    3.352e+11     9.107e+11     4.163e+11


Final Statistics..

lenrw   =  2089     leniw   =    50
lenrwLS =  2450     leniwLS =    22
nst     =   491
nfe     =   631     nfeLS   =     0
nni     =   627     nli     =   643
nsetups =    82     netf    =    29
npe     =     9     nps     =  1214
ncfn    =     0     ncfl    =     0
```

3 Parallel example problems

3.1 A nonstiff example: cvAdvDiff_non_p

This problem begins with a simple diffusion-advection equation for $u = u(t, x)$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0.5\frac{\partial u}{\partial x} \tag{7}$$

for $0 \le t \le 5$, $0 \le x \le 2$, and subject to homogeneous Dirichlet boundary conditions and initial values given by

$$\begin{aligned} u(t,0) &= 0\ , \quad u(t,2) = 0\ , \\ u(0,x) &= x(2-x)e^{2x}\ . \end{aligned} \tag{8}$$

A system of `MX` ODEs is obtained by discretizing the x-axis with `MX`+2 grid points and replacing the first and second order spatial derivatives with their central difference approximations. Since the value of u is constant at the two endpoints, the semi-discrete equations for those points can be eliminated. With u_i as the approximation to $u(t, x_i)$, $x_i = i(\Delta x)$, and $\Delta x = 2/(\texttt{MX}+1)$, the resulting system of ODEs, $\dot{u} = f(t, u)$, can now be written:

$$\dot{u}_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + 0.5\frac{u_{i+1} - u_{i-1}}{2(\Delta x)}\ . \tag{9}$$

This equation holds for $i = 1, 2, \ldots,$ `MX`, with the understanding that $u_0 = u_{\texttt{MX}+1} = 0$.

In the parallel processing environment, we may think of the several processors as being laid out on a straight line with each processor to compute its contiguous subset of the solution vector. Consequently the computation of the right hand side of Eq. (9) requires that each interior processor must pass the first component of its block of the solution vector to its left-hand neighbor, acquire the last component of that neighbor's block, pass the last component of its block of the solution vector to its right-hand neighbor, and acquire the first component of that neighbor's block. If the processor is the first (0th) or last processor, then communication to the left or right (respectively) is not required.

This problem uses the Adams (non-stiff) integration formula and functional iteration. It is unrealistically simple, but serves to illustrate use of the parallel version of CVODE.

The `cvAdvDiff_non_p.c` file begins with `#include` declarations for various required header files, including lines for `nvector_parallel` to access the parallel `N_Vector` type and related macros, and for `mpi.h` to access MPI types and constants. Following that are definitions of problem constants and a data block for communication with the `f` routine. That block includes the number of PEs, the index of the local PE, and the MPI communicator.

The `main` program begins with MPI calls to initialize MPI and to set multi-processor environment parameters `npes` (number of PEs) and `my_pe` (local PE index). The local vector length is set according to `npes` and the problem size `NEQ` (which may or may not be multiple of `npes`). The value `my_base` is the base value for computing global indices (from 1 to `NEQ`) for the local vectors. The solution vector `u` is created with a call to `N_VNew_Parallel` and loaded with a call to `SetIC`. The calls to `CVodeCreate`, `CVodeInit`, and `CVodeSStolerances` specify a CVODE solution with the nonstiff method and scalar tolerances. The call to `CVodeSetUserdata` insures that the pointer `data` is passed to the `f` routine whenever it is called. A heading is printed (if on processor 0). In a loop over `tout` values, `CVode` is called, and the return value checked for errors. The max-norm of the solution and the total number

of time steps so far are printed at each output point. Finally, some statistical counters are printed, memory is freed, and MPI is finalized.

The `SetIC` routine uses the last two arguments passed to it to compute the set of global indices (`my_base`+1 to `my_base+my_length`) corresponding to the local part of the solution vector `u`, and then to load the corresponding initial values. The `PrintFinalStats` routine uses `CVodeGet***` calls to get various counters, and then prints these. The counters are: `nst` (number of steps), `nfe` (number of `f` evaluations), `nni` (number of nonlinear iterations), `netf` (number of error test failures), and `ncfn` (number of nonlinear convergence failures). This routine is suitable for general use with CVODE applications to nonstiff problems.

The `f` function is an implementation of Eq. (9), but preceded by communication operations appropriate for the parallel setting. It copies the local vector `u` into a larger array `z`, shifted by 1 to allow for the storage of immediate neighbor components. The first and last components of `u` are sent to neighboring processors with `MPI_Send` calls, and the immediate neighbor solution values are received from the neighbor processors with `MPI_Recv` calls, except that zero is loaded into `z[0]` or `z[my_length+1]` instead if at the actual boundary. Then the central difference expressions are easily formed from the `z` array, and loaded into the data array of the `udot` vector.

The `cvAdvDiff_non_p.c` file includes a routine `check_flag` that checks the return values from calls in `main`. This routine was written to be used by any parallel SUNDIALS application.

The output below is for `cvAdvDiff_non_p` with `MX` = 10 and four processors. Varying the number of processors will alter the output, only because of roundoff-level differences in various vector operations. The fairly high value of `ncfn` indicates that this problem is on the borderline of being stiff.

cvAdvDiff_non_p sample output

```
 1-D advection-diffusion equation, mesh size = 10

 Number of PEs =   2

At t = 0.00   max.norm(u) =  1.569909e+01  nst =   0
At t = 0.50   max.norm(u) =  3.052881e+00  nst = 113
At t = 1.00   max.norm(u) =  8.753188e-01  nst = 191
At t = 1.50   max.norm(u) =  2.494926e-01  nst = 265
At t = 2.00   max.norm(u) =  7.109437e-02  nst = 332
At t = 2.50   max.norm(u) =  2.026176e-02  nst = 405
At t = 3.00   max.norm(u) =  5.769193e-03  nst = 468
At t = 3.50   max.norm(u) =  1.651582e-03  nst = 541
At t = 4.00   max.norm(u) =  4.764197e-04  nst = 623
At t = 4.50   max.norm(u) =  1.370219e-04  nst = 704
At t = 5.00   max.norm(u) =  3.990405e-05  nst = 785

Final Statistics:

nst = 785     nfe  = 1442    nni = 1438    ncfn = 140     netf = 5
```

3.2 A user preconditioner example: cvDiurnal_kry_p

As an example of using CVODE with the Krylov linear solver SUNLINSOL_SPGMR, iterative linear solver interfce CVSPILS, and the parallel MPI NVECTOR_PARALLEL module, we describe

a test problem based on the system PDEs given above for the `cvDiurnal_kry` example. As before, we discretize the PDE system with central differencing, to obtain an ODE system $\dot{u} = f(t, u)$ representing (4). But in this case, the discrete solution vector is distributed over many processors. Specifically, we may think of the processors as being laid out in a rectangle, and each processor being assigned a subgrid of size `MXSUB`×`MYSUB` of the $x - y$ grid. If there are `NPEX` processors in the x direction and `NPEY` processors in the y direction, then the overall grid size is `MX`×`MY` with `MX`=`NPEX`×`MXSUB` and `MY`=`NPEY`×`MYSUB`, and the size of the ODE system is 2·`MX`·`MY`.

To compute f in this setting, the processors pass and receive information as follows. The solution components for the bottom row of grid points in the current processor are passed to the processor below it and the solution for the top row of grid points is received from the processor below the current processor. The solution for the top row of grid points for the current processor is sent to the processor above the current processor, while the solution for the bottom row of grid points is received from that processor by the current processor. Similarly the solution for the first column of grid points is sent from the current processor to the processor to its left and the last column of grid points is received from that processor by the current processor. The communication for the solution at the right edge of the processor is similar. If this is the last processor in a particular direction, then message passing and receiving are bypassed for that direction.

This code is intended to provide a more realistic example than that in `cvAdvDiff_non_p`, and to provide a template for a stiff ODE system arising from a PDE system. The solution method is BDF with Newton iteration and SPGMR. The left preconditioner is the block-diagonal part of the Newton matrix, with 2×2 blocks, and the corresponding diagonal blocks of the Jacobian are saved each time the preconditioner is generated, for re-use later under certain conditions.

The organization of the `cvDiurnal_kry_p` program deserves some comments. The right-hand side routine `f` calls two other routines: `ucomm`, which carries out inter-processor communication; and `fcalc`, which operates on local data only and contains the actual calculation of $f(t, u)$. The `ucomm` function in turn calls three routines which do, respectively, non-blocking receive operations, blocking send operations, and receive-waiting. All three use MPI, and transmit data from the local `u` vector into a local working array `uext`, an extended copy of `u`. The `fcalc` function copies `u` into `uext`, so that the calculation of $f(t, u)$ can be done conveniently by operations on `uext` only. Most other features of `cvDiurnal_kry_p.c` are the same as in `cvDiurnal_kry.c`, except for extra logic involved with distributed vectors.

The following is a sample output from `cvDiurnal_kry_p`, for four processors (in a 2×2 array) with a 5×5 subgrid on each. The output will vary slightly if the number of processors is changed.

cvDiurnal_kry_p sample output

```
2-species diurnal advection-diffusion problem

t = 7.20e+03   no. steps = 219   order = 5   stepsize = 1.59e+02
At bottom left:  c1, c2 =    1.047e+04    2.527e+11
At top right:    c1, c2 =    1.119e+04    2.700e+11

t = 1.44e+04   no. steps = 251   order = 5   stepsize = 3.77e+02
At bottom left:  c1, c2 =    6.659e+06    2.582e+11
At top right:    c1, c2 =    7.301e+06    2.833e+11
```

```
t = 2.16e+04   no. steps = 277   order = 5   stepsize = 2.75e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

t = 2.88e+04   no. steps = 321   order = 3   stepsize = 5.19e+01
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 384   order = 4   stepsize = 8.70e+01
At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 456   order = 4   stepsize = 8.77e+02
At bottom left:  c1, c2 =    1.077e-06    3.382e+11
At top right:    c1, c2 =    4.057e-07    3.804e+11

t = 5.04e+04   no. steps = 471   order = 4   stepsize = 3.29e+02
At bottom left:  c1, c2 =   -1.176e-08    3.358e+11
At top right:    c1, c2 =   -5.053e-08    3.864e+11

t = 5.76e+04   no. steps = 488   order = 5   stepsize = 3.95e+02
At bottom left:  c1, c2 =   -9.464e-11    3.320e+11
At top right:    c1, c2 =   -3.493e-10    3.909e+11

t = 6.48e+04   no. steps = 501   order = 5   stepsize = 6.20e+02
At bottom left:  c1, c2 =    5.057e-11    3.313e+11
At top right:    c1, c2 =    1.868e-10    3.963e+11

t = 7.20e+04   no. steps = 512   order = 5   stepsize = 6.20e+02
At bottom left:  c1, c2 =   -4.454e-11    3.330e+11
At top right:    c1, c2 =   -1.629e-10    4.039e+11

t = 7.92e+04   no. steps = 524   order = 5   stepsize = 6.20e+02
At bottom left:  c1, c2 =   -2.189e-13    3.334e+11
At top right:    c1, c2 =   -8.112e-13    4.120e+11

t = 8.64e+04   no. steps = 535   order = 5   stepsize = 6.20e+02
At bottom left:  c1, c2 =    1.080e-15    3.352e+11
At top right:    c1, c2 =    3.729e-15    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2450     leniwls =   106
nst     =   535
nfe     =   688     nfels   =   668
nni     =   684     nli     =   668
nsetups =    92     netf    =    33
npe     =    10     nps     =  1294
ncfn    =     0     ncfl    =     1
```

3.3 A CVBBDPRE preconditioner example: cvDiurnal_kry_bbd_p

In this example, `cvDiurnal_kry_bbd_p`, we solve the same problem as in `cvDiurnal_kry_p` above, but instead of supplying the preconditioner, we use the CVBBDPRE module, which

generates and uses a band-block-diagonal preconditioner. The half-bandwidths of the Jacobian block on each processor are both equal to 2·MXSUB, and that is the value supplied as mudq and mldq in the call to CVBBDPrecInit. But in order to reduce storage and computation costs for preconditioning, we supply the values mukeep = mlkeep = 2 (= NVARS) as the half-bandwidths of the retained band matrix blocks. This means that the Jacobian elements are computed with a difference quotient scheme using the true bandwidth of the block, but only a narrow band matrix (bandwidth 5) is kept as the preconditioner.

As in cvDiurnal_kry_p.c, the f routine in cvDiurnal_kry_bbd_p.c simply calls a communication routine, fucomm, and then a strictly computational routine, flocal. However, the call to CVBBDPrecInit specifies the pair of routines to be called as ucomm and flocal, where ucomm is NULL. This is because each call by the solver to ucomm is preceded by a call to f with the same (t,u) arguments, and therefore the communication needed for flocal in the solver's calls to it have already been done.

In cvDiurnal_kry_bbd_p.c, the problem is solved twice — first with preconditioning on the left, and then on the right. Thus prior to the second solution, calls are made to reset the initial values (SetInitialProfiles), the main solver memory (CVodeReInit), the CVBBDPRE memory (CVBBDPrecReInit), as well as the preconditioner type (SUNSPGMRSetPrecType).

Sample output from cvDiurnal_kry_bbd_p follows, again using 5×5 subgrids on a 2×2 processor grid. The performance of the preconditioner, as measured by the number of Krylov iterations per Newton iteration, nli/nni, is very close to that of cvDiurnal_kry_p when preconditioning is on the left, but slightly poorer when it is on the right.

cvDiurnal_kry_bbd_p sample output

```
2-species diurnal advection-diffusion problem
  10 by 10 mesh on 4 processors
  Using CVBBDPRE preconditioner module
    Difference-quotient half-bandwidths are mudq = 10,  mldq = 10
    Retained band block half-bandwidths are mukeep = 2,  mlkeep = 2

Preconditioner type is:  jpre = PREC_LEFT

t = 7.20e+03   no. steps = 190   order = 5   stepsize = 1.61e+02
At bottom left:  c1, c2 =    1.047e+04    2.527e+11
At top right:    c1, c2 =    1.119e+04    2.700e+11

t = 1.44e+04   no. steps = 221   order = 5   stepsize = 3.85e+02
At bottom left:  c1, c2 =    6.659e+06    2.582e+11
At top right:    c1, c2 =    7.301e+06    2.833e+11

t = 2.16e+04   no. steps = 247   order = 5   stepsize = 3.00e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

t = 2.88e+04   no. steps = 272   order = 4   stepsize = 2.13e+02
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 311   order = 4   stepsize = 9.70e+01
At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 368   order = 4   stepsize = 3.95e+02
At bottom left:  c1, c2 =    6.243e-08    3.382e+11
```

```
At top right:    c1, c2 =    7.021e-08    3.804e+11

t = 5.04e+04   no. steps = 388   order = 5   stepsize = 4.74e+02
At bottom left:  c1, c2 =    3.000e-07    3.358e+11
At top right:    c1, c2 =    3.306e-07    3.864e+11

t = 5.76e+04   no. steps = 401   order = 5   stepsize = 3.61e+02
At bottom left:  c1, c2 =   -1.096e-10    3.320e+11
At top right:    c1, c2 =   -6.268e-11    3.909e+11

t = 6.48e+04   no. steps = 414   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =    1.186e-11    3.313e+11
At top right:    c1, c2 =    6.568e-14    3.963e+11

t = 7.20e+04   no. steps = 425   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =   -7.713e-12    3.330e+11
At top right:    c1, c2 =    5.432e-13    4.039e+11

t = 7.92e+04   no. steps = 436   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =    2.525e-13    3.334e+11
At top right:    c1, c2 =   -2.072e-14    4.120e+11

t = 8.64e+04   no. steps = 448   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =    4.758e-15    3.352e+11
At top right:    c1, c2 =    7.572e-17    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2450     leniwls =   106
nst     =   448
nfe     =   581     nfels   =   509
nni     =   577     nli     =   509
nsetups =    76     netf    =    26
npe     =     8     nps     =  1029
ncfn    =     0     ncfl    =     0

In CVBBDPRE: real/integer local work space sizes = 1300, 192
             no. flocal evals. = 176


-------------------------------------------------------------------


Preconditioner type is:  jpre = PREC_RIGHT

t = 7.20e+03   no. steps = 191   order = 5   stepsize = 1.22e+02
At bottom left:  c1, c2 =    1.047e+04    2.527e+11
At top right:    c1, c2 =    1.119e+04    2.700e+11

t = 1.44e+04   no. steps = 223   order = 5   stepsize = 2.79e+02
At bottom left:  c1, c2 =    6.659e+06    2.582e+11
At top right:    c1, c2 =    7.301e+06    2.833e+11

t = 2.16e+04   no. steps = 249   order = 5   stepsize = 4.31e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11
```

```
t = 2.88e+04   no. steps = 309   order = 4   stepsize = 1.32e+02
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 342   order = 5   stepsize = 1.28e+02
At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 393   order = 4   stepsize = 3.91e+02
At bottom left:  c1, c2 =    1.998e-09    3.382e+11
At top right:    c1, c2 =    2.210e-09    3.804e+11

t = 5.04e+04   no. steps = 406   order = 5   stepsize = 6.68e+02
At bottom left:  c1, c2 =    4.173e-11    3.358e+11
At top right:    c1, c2 =    4.509e-11    3.864e+11

t = 5.76e+04   no. steps = 421   order = 4   stepsize = 1.46e+02
At bottom left:  c1, c2 =    1.346e-13    3.320e+11
At top right:    c1, c2 =    1.429e-13    3.909e+11

t = 6.48e+04   no. steps = 436   order = 4   stepsize = 5.90e+02
At bottom left:  c1, c2 =    4.188e-18    3.313e+11
At top right:    c1, c2 =   -4.796e-15    3.963e+11

t = 7.20e+04   no. steps = 448   order = 4   stepsize = 5.90e+02
At bottom left:  c1, c2 =   -4.919e-18    3.330e+11
At top right:    c1, c2 =    8.770e-17    4.039e+11

t = 7.92e+04   no. steps = 460   order = 4   stepsize = 5.90e+02
At bottom left:  c1, c2 =    2.462e-20    3.334e+11
At top right:    c1, c2 =   -2.599e-20    4.120e+11

t = 8.64e+04   no. steps = 472   order = 4   stepsize = 5.90e+02
At bottom left:  c1, c2 =    3.021e-23    3.352e+11
At top right:    c1, c2 =   -3.233e-23    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2450     leniwls =   106
nst     =   472
nfe     =   604     nfels   =   758
nni     =   600     nli     =   758
nsetups =    94     netf    =    29
npe     =     9     nps     =  1257
ncfn    =     0     ncfl    =     0

In CVBBDPRE: real/integer local work space sizes = 1300, 192
             no. flocal evals. = 198
```

4 *hypre* example problems

4.1 A nonstiff example: cvAdvDiff_non_ph

This example is same as `cvAdvDiff_non_p`, except that it uses the *hypre* vector type instead of the SUNDIALS native parallel vector implementation. The outputs from the two examples are identical. In the following, we will point out only the differences between the two. Familiarity with *hypre* library [1] is helpful.

We use the *hypre* IJ vector interface to allocate the template vector and create parallel partitioning:

```
HYPRE_IJVectorCreate(comm, my_base, my_base + local_N - 1, &Uij);
HYPRE_IJVectorSetObjectType(Uij, HYPRE_PARCSR);
HYPRE_IJVectorInitialize(Uij);
```

The initialize call means that vector elements are ready to be set using the IJ interface. We choose an initial condition vector $x_0 = x(t_0)$ as the template vector and we set its values in the `SetIC(...)` function. We complete the *hypre* vector assembly by:

```
HYPRE_IJVectorAssemble(Uij);
HYPRE_IJVectorGetObject(Uij, (void**) &Upar);
```

The assemble call is collective and it makes *hypre* vector ready to use. This sets the handle `Upar` to the actual *hypre* vector. The handle is then passed to the `N_VMake` function, which creates the template `N_Vector` as a wrapper around the *hypre* vector. All other vectors in the computation are created by cloning the template vector. The template vector does not own the underlying *hypre* vector, and it is the user's responsibility to destroy it using a `HYPRE_IJVectorDestroy(Uij)` call after the template vector has been destroyed. This function will destroy both the *hypre* vector and its IJ interface.

To access individual elements of solution vectors `u` and `udot` in the residual function, the user needs to extract the *hypre* vector first by calling `N_VGetVector_ParHyp`, and then use *hypre* methods from that point on.

Notes

- At this point interfaces to *hypre* solvers and preconditioners are not available. They will be provided in subsequent SUNDIALS releases. The interface to *hypre* vector is included in this release mainly for testing purposes and as a preview of functionality to come.

5 CUDA example problems

5.1 An unpreconditioned Krylov example: cvAdvDiff_kry_cuda

The example program `cvAdvDiff_kry_cuda.cu` solves the same 2-D advection-diffusion equation as in Section 2.2, but instead of using banded direct solver, it uses unpreconditioned Krylov solver. Here we only highlight differences between the two examples.

The `cvAdvDiff_kry_cuda.cu` program includes files `sunlinsol_spmgr.h` in order to use the SPGMR Krylov linear solver. File `cvode_spils.h` provides prototypes for `CVSpilsSetLinearSolver`, which sets the iterative linear solver for CVODE, and `CVSpilsSetJacTimes`, which sets the pointer to user supplied Jacobian-vector product function. The file `nvector_cuda.h` is included for the definition of the CUDA `N_Vector` type. The prototype vector is created using `N_VNew_Cuda` function.

In order to get a good performance and avoid moving data between host and device at every iteration, it is recommended that user evaluates model at the device. In the example, model right hand side and Jacobian-vector product are implemented as CUDA kernels `fKernel` and `jtvKernel`, respectively. User provided C functions `f` and `jtv`, which are called directly by CVODE, set thread partitioning and launch thier respective CUDA kernels. Vector data on the device is accessed using `N_VGetDeviceArrayPointer_Cuda` function.

The output generated by `cvAdvDiff_kry_cuda` is shown below.

cvAdvDiff_kry_cuda sample output

```
2-D Advection-Diffusion Equation
Mesh dimensions = 10 X 5
Total system size = 50
Tolerance parameters: reltol = 0   abstol = 1e-05

At t = 0      max.norm(u) =  8.954716e+01
At t = 0.10   max.norm(u) =  4.132884e+00   nst =   82
At t = 0.20   max.norm(u) =  1.039293e+00   nst =  102
At t = 0.30   max.norm(u) =  2.979816e-01   nst =  112
At t = 0.40   max.norm(u) =  8.765538e-02   nst =  119
At t = 0.50   max.norm(u) =  2.625408e-02   nst =  125
At t = 0.60   max.norm(u) =  7.826077e-03   nst =  129
At t = 0.70   max.norm(u) =  2.326537e-03   nst =  133
At t = 0.80   max.norm(u) =  6.891180e-04   nst =  137
At t = 0.90   max.norm(u) =  2.039853e-04   nst =  140
At t = 1.00   max.norm(u) =  5.925586e-05   nst =  143

Final Statistics..

nst     =   143
nfe     =   207     nfeLS   =     0
nni     =   203     nli     =   225
nsetups =     0     netf    =     2
npe     =     0     nps     =     0
ncfn    =     0     ncfl    =     0
```

6 RAJA example problems

6.1 An unpreconditioned Krylov example: cvAdvDiff_kry_raja

The example program `cvAdvDiff_kry_raja.cu` solves the same 2-D advection-diffusion equation as in Sections 2.2 and 5.1.

The file `nvector_raja.h` contains the definition of the RAJA `N_Vector` type, and `RAJA.hpp` definition of the RAJA `forall` loops. The prototype vector in the main body of the program is created using `N_VNew_Raja` function.

In order to get a good performance and avoid moving data between host and device at every iteration, it is recommended that user evaluates model at the device. In the example, user-supplied model right hand side and Jacobian-vector product functions, `f` and `jtv`, operate on the device data. Vector data on the device is accessed using `N_VGetDeviceArrayPointer_Raja` function. Looping over vector components is implemented using RAJA `forall` loops.

The output generated by `cvAdvDiff_kry_raja` is shown below.

cvAdvDiff_kry_raja sample output

```
2-D Advection-Diffusion Equation
Mesh dimensions = 10 X 5
Total system size = 50
Tolerance parameters: reltol = 0   abstol = 1e-05

At t = 0      max.norm(u) =  8.954716e+01
At t = 0.10   max.norm(u) =  4.132884e+00   nst =   82
At t = 0.20   max.norm(u) =  1.039293e+00   nst =  102
At t = 0.30   max.norm(u) =  2.979816e-01   nst =  112
At t = 0.40   max.norm(u) =  8.765538e-02   nst =  119
At t = 0.50   max.norm(u) =  2.625408e-02   nst =  125
At t = 0.60   max.norm(u) =  7.826077e-03   nst =  129
At t = 0.70   max.norm(u) =  2.326537e-03   nst =  133
At t = 0.80   max.norm(u) =  6.891180e-04   nst =  137
At t = 0.90   max.norm(u) =  2.039853e-04   nst =  140
At t = 1.00   max.norm(u) =  5.925586e-05   nst =  143

Final Statistics..

nst     =   143
nfe     =   207     nfeLS   =     0
nni     =   203     nli     =   225
nsetups =     0     netf    =     2
npe     =     0     nps     =     0
ncfn    =     0     ncfl    =     0
```

7 Fortran example problems

The FORTRAN example problem programs supplied with the CVODE package are all written in standard FORTRAN77 and use double precision arithmetic. Before running any of these examples, the user should make sure that the FORTRAN data types for real and integer variables appropriately match the C types. See §5.2 in the CVODE User Document for details.

7.1 A serial example: fcvDiurnal_kry

The `fcvDiurnal_kry` example is a FORTRAN equivalent of the `cvDiurnal_kry` problem. (In fact, it was derived from an earlier FORTRAN example program for VODPK.)

The main program begins with a call to `INITKX`, which sets problem parameters, loads these into arrays `IPAR` and `RPAR` for use by other routines, and loads `Y` (here called `U0`) with its initial values. `Main` calls `FNVINITS`, `FSUNSPGMRINIT`, `FSUNSPGMRSETGSTYPE`, `FCVMALLOC`, `FCVSPILSINIT`, and `FCVSPILSSETPREC`, to initialize the NVECTOR_SERIAL module, the SUNLINSOL_SPGMR module, the main solver memory, and the CVSPILS interface, and to specify user-supplied preconditioner setup and solve routines. It calls `FCVODE` in a loop over `TOUT` values, with printing of selected solution values and performance data (from the `IOUT` and `ROUT` arrays). At the end, it prints a number of performance counters, and frees memory with calls to `FCVFREE`.

In `fcvDiurnal_kry.f`, the `FCVFUN` routine is a straightforward implementation of the discretized form of Eqns. (4). In `FCVPSET`, the block-diagonal part of the Jacobian, J_{bd}, is computed (and copied to `P`) if `JOK` $= 0$, but is simply copied from `BD` to `P` if `JOK` $= 1$. In both cases, the preconditioner matrix P is formed from J_{bd} and its 2×2 blocks are LU-factored. In `FCVPSOL`, the solution of a linear system $Px = z$ is solved by doing backsolve operations on the blocks. Subordinate routines are used to isolate these evaluation, factorization, and backsolve operations. The remainder of `fcvDiurnal_kry.f` consists of routines from LINPACK and the BLAS needed for matrix and vector operations.

The following is sample output from `fcvDiurnal_kry`, using a 10×10 mesh. The performance of FCVODE here is quite similar to that of CVODE on the `cvDiurnal_kry` problem, as expected.

fcvDiurnal_kry sample output

```
Krylov example problem:

 Kinetics-transport, NEQ =  200


 t =   0.720E+04   nst =   219  q =  5  h =   0.158696E+03
  c1 (bot.left/middle/top rt.) =   0.104683E+05  0.296373E+05  0.111853E+05
  c2 (bot.left/middle/top rt.) =   0.252672E+12  0.715376E+12  0.269977E+12

 t =   0.144E+05   nst =   251  q =  5  h =   0.377205E+03
  c1 (bot.left/middle/top rt.) =   0.665902E+07  0.531602E+07  0.730081E+07
  c2 (bot.left/middle/top rt.) =   0.258192E+12  0.205680E+12  0.283286E+12

 t =   0.216E+05   nst =   277  q =  5  h =   0.274584E+03
  c1 (bot.left/middle/top rt.) =   0.266498E+08  0.103636E+08  0.293077E+08
  c2 (bot.left/middle/top rt.) =   0.299279E+12  0.102810E+12  0.331344E+12

 t =   0.288E+05   nst =   307  q =  4  h =   0.201171E+03
  c1 (bot.left/middle/top rt.) =   0.870209E+07  0.129197E+08  0.965002E+07
```

```
  c2 (bot.left/middle/top rt.) =   0.338035E+12  0.502929E+12  0.375096E+12

 t =   0.360E+05   nst =   336  q =  5  h =   0.101863E+03
  c1 (bot.left/middle/top rt.) =   0.140404E+05  0.202903E+05  0.156090E+05
  c2 (bot.left/middle/top rt.) =   0.338677E+12  0.489443E+12  0.376516E+12

 t =   0.432E+05   nst =   385  q =  4  h =   0.456580E+03
  c1 (bot.left/middle/top rt.) =   0.522615E-07 -0.607462E-05  0.871309E-07
  c2 (bot.left/middle/top rt.) =   0.338233E+12  0.135487E+12  0.380352E+12

 t =   0.504E+05   nst =   403  q =  4  h =   0.277279E+03
  c1 (bot.left/middle/top rt.) =  -0.266960E-07 -0.450968E-05 -0.267202E-07
  c2 (bot.left/middle/top rt.) =   0.335816E+12  0.493033E+12  0.386445E+12

 t =   0.576E+05   nst =   419  q =  5  h =   0.344023E+03
  c1 (bot.left/middle/top rt.) =  -0.114670E-08 -0.587145E-07 -0.219472E-08
  c2 (bot.left/middle/top rt.) =   0.332031E+12  0.964981E+12  0.390900E+12

 t =   0.648E+05   nst =   431  q =  5  h =   0.949533E+03
  c1 (bot.left/middle/top rt.) =   0.379212E-09  0.190141E-07  0.719619E-09
  c2 (bot.left/middle/top rt.) =   0.331303E+12  0.892184E+12  0.396342E+12

 t =   0.720E+05   nst =   441  q =  5  h =   0.700864E+03
  c1 (bot.left/middle/top rt.) =  -0.187184E-10 -0.936079E-09 -0.355015E-10
  c2 (bot.left/middle/top rt.) =   0.332972E+12  0.618617E+12  0.403885E+12

 t =   0.792E+05   nst =   451  q =  5  h =   0.700864E+03
  c1 (bot.left/middle/top rt.) =   0.136509E-12  0.673869E-11  0.258515E-12
  c2 (bot.left/middle/top rt.) =   0.333441E+12  0.666904E+12  0.412026E+12

 t =   0.864E+05   nst =   461  q =  5  h =   0.700864E+03
  c1 (bot.left/middle/top rt.) =   0.414351E-13  0.207247E-11  0.786247E-13
  c2 (bot.left/middle/top rt.) =   0.335178E+12  0.910668E+12  0.416250E+12


Final statistics:

 number of steps        =   461      number of f evals.     =  598
 number of prec. setups =    78
 number of prec. evals. =     8      number of prec. solves =  1176
 number of nonl. iters. =   594      number of lin. iters.  =   636
 average Krylov subspace dimension (NLI/NNI)  =   0.107071E+01
 number of conv. failures.. nonlinear =   0 linear =   0
 number of error test failures =  27
```

7.2 A parallel example: fcvDiag_kry_bbd_p

This example, `fcvDiag_kry_bbd_p`, uses a simple diagonal ODE system to illustrate the use of FCVODE in a parallel setting. The system is

$$\dot{y}_i = -\alpha\ i\ y_i \quad (i = 1, \ldots, N) \tag{10}$$

on the time interval $0 \leq t \leq 1$. In this case, we use $\alpha = 10$ and $N = 10{*}$`NPES`, where `NPES` is the number of processors and is specified at run time. The linear solver to be used is SPGMR with the CVBBDPRE (band-block-diagonal) preconditioner. Since the system Jacobian is diagonal, the half-bandwidths specified are all zero. The problem is solved twice — with preconditioning on the left, then on the right.

The source file for this problem begins with MPI calls to initialize MPI and to get the number of processors and local processor index. FNVINITP is called to initialize the MPI-parallel NVECTOR module, while FSUNSPGMRINIT and FSUNSPGMRSETGSTYPE are called to initialize the SPGMR SUNLINSOL module. Following the call to FCVMALLOC, the linear solver and preconditioner are attached to CVODE with calls to FCVSPILSINIT and FCVBBDINIT. In a loop over TOUT values, it calls FCVODE and prints the step and f evaluation counters. After that, it computes and prints the maximum global error, and all the relevant performance counters. Those specific to CVBBDPRE are obtained by a call to FCVBBDOPT. To prepare for the second run, the program calls FCVREINIT, FCVBBDREINIT, and FSUNSPGMRSETPRECTYPE, in addition to resetting the initial conditions. Finally, it frees memory and terminates MPI. Notice that in the FCVFUN routine, the local processor index MYPE and the local vector size NLOCAL are used to form the global index values needed to evaluate the right-hand side of Eq. (10).

The following is a sample output from fcvDiag_kry_bbd_p, with NPES = 4. As expected, the performance is identical for left vs right preconditioning.

fcvDiag_kry_bbd_p sample output

```
Diagonal test problem:

 NEQ =  20
 parameter alpha =   10.000
 ydot_i = -alpha*i * y_i (i = 1,...,NEQ)
 RTOL, ATOL =    0.1E-04   0.1E-09
 Method is BDF/NEWTON/SPGMR
 Preconditioner is band-block-diagonal, using CVBBDPRE
 Number of processors =   2


Preconditioning on left

 t =   0.10E+00     no. steps =   174   no. f-s =   214
 t =   0.20E+00     no. steps =   222   no. f-s =   263
 t =   0.30E+00     no. steps =   247   no. f-s =   289
 t =   0.40E+00     no. steps =   265   no. f-s =   308
 t =   0.50E+00     no. steps =   278   no. f-s =   322
 t =   0.60E+00     no. steps =   290   no. f-s =   334
 t =   0.70E+00     no. steps =   300   no. f-s =   345
 t =   0.80E+00     no. steps =   307   no. f-s =   352
 t =   0.90E+00     no. steps =   312   no. f-s =   358
 t =   0.10E+01     no. steps =   317   no. f-s =   363

Max. absolute error is  0.90E-08


Final statistics:

 number of steps        =   317     number of f evals.     =   363
 number of prec. setups =    33
 number of prec. evals. =     6     number of prec. solves =   644
 number of nonl. iters. =   359     number of lin. iters.  =   322
 average Krylov subspace dimension (NLI/NNI) =   0.8969
 number of conv. failures.. nonlinear =   0  linear =   0
 number of error test failures =   5
 main solver real/int workspace sizes   =   289   80
 linear solver real/int workspace sizes =   290   58
```

```
In CVBBDPRE:
 real/int local workspace =   100   60
 number of g evals. =    12


------------------------------------------------------------


Preconditioning on right

 t =   0.10E+00     no. steps =   174   no. f-s =   214
 t =   0.20E+00     no. steps =   222   no. f-s =   263
 t =   0.30E+00     no. steps =   247   no. f-s =   289
 t =   0.40E+00     no. steps =   265   no. f-s =   308
 t =   0.50E+00     no. steps =   278   no. f-s =   322
 t =   0.60E+00     no. steps =   290   no. f-s =   334
 t =   0.70E+00     no. steps =   300   no. f-s =   345
 t =   0.80E+00     no. steps =   307   no. f-s =   352
 t =   0.90E+00     no. steps =   312   no. f-s =   358
 t =   0.10E+01     no. steps =   317   no. f-s =   363

Max. absolute error is  0.90E-08


Final statistics:

 number of steps        =   317     number of f evals.     =   363
 number of prec. setups =    33
 number of prec. evals. =     6     number of prec. solves =   644
 number of nonl. iters. =   359     number of lin. iters.  =   322
 average Krylov subspace dimension (NLI/NNI) =   0.8969
 number of conv. failures.. nonlinear =   0  linear =   0
 number of error test failures =   5
 main solver real/int workspace sizes   =   289   80
 linear solver real/int workspace sizes =   290   58
In CVBBDPRE:
 real/int local workspace =   100   60
 number of g evals. =    12
```

8 Parallel tests

The stiff example problem `cvDiurnal_kry` described above, or rather its parallel version `cvDiurnal_kry_p`, has been modified and expanded to form a test problem for the parallel version of CVODE. This work was largely carried out by M. Wittman and reported in [3].

To start with, in order to add realistic complexity to the solution, the initial profile for this problem was altered to include a rather steep front in the vertical direction. Specifically, the function $\beta(y)$ in Eq. (6) has been replaced by:

$$\beta(y) = .75 + .25 \tanh(10y - 400) \ . \tag{11}$$

This function rises from about .5 to about 1.0 over a y interval of about .2 (i.e. 1/100 of the total span in y). This vertical variation, together with the horizonatal advection and diffusion in the problem, demands a fairly fine spatial mesh to achieve acceptable resolution.

In addition, an alternate choice of differencing is used in order to control spurious oscillations resulting from the horizontal advection. In place of central differencing for that term, a biased upwind approximation is applied to each of the terms $\partial c^i/\partial x$, namely:

$$\partial c/\partial x|_{x_j} \approx \left[\frac{3}{2}c_{j+1} - c_j - \frac{1}{2}c_{j-1}\right] /(2\Delta x) \ . \tag{12}$$

With this modified form of the problem, we performed tests similar to those described above for the example. Here we fix the subgrid dimensions at `MXSUB = MYSUB =` 50, so that the local (per-processor) problem size is 5000, while the processor array dimensions, `NPEX` and `NPEY`, are varied. In one (typical) sequence of tests, we fix `NPEY` = 8 (for a vertical mesh size of `MY` = 400), and set `NPEX` = 8 (`MX` = 400), `NPEX` = 16 (`MX` = 800), and `NPEX` = 32 (`MX` = 1600). Thus the largest problem size N is $2 \cdot 400 \cdot 1600 = 1,280,000$. For these tests, we also raise the maximum Krylov dimension, `maxl`, to 10 (from its default value of 5).

For each of the three test cases, the test program was run on a Cray-T3D (256 processors) with each of three different message-passing libraries:

- MPICH: an implementation of MPI on top of the Chameleon library
- EPCC: an implementation of MPI by the Edinburgh Parallel Computer Centre
- SHMEM: Cray's Shared Memory Library

The following table gives the run time and selected performance counters for these 9 runs. In all cases, the solutions agreed well with each other, showing expected small variations with grid size. In the table, M-P denotes the message-passing library, RT is the reported run time in CPU seconds, `nst` is the number of time steps, `nfe` is the number of f evaluations, `nni` is the number of nonlinear (Newton) iterations, `nli` is the number of linear (Krylov) iterations, and `npe` is the number of evaluations of the preconditioner.

Some of the results were as expected, and some were surprising. For a given mesh size, variations in performance counts were small or absent, except for moderate (but still acceptable) variations for SHMEM in the smallest case. The increase in costs with mesh size can be attributed to a decline in the quality of the preconditioner, which neglects most of the spatial coupling. The preconditioner quality can be inferred from the ratio `nli/nni`, which is the average number of Krylov iterations per Newton iteration. The most interesting (and unexpected) result is the variation of run time with library: SHMEM is the most efficient,

NPEX	M-P	RT	nst	nfe	nni	nli	npe
8	MPICH	436.	1391	9907	1512	8392	24
8	EPCC	355.	1391	9907	1512	8392	24
8	SHMEM	349.	1999	10,326	2096	8227	34
16	MPICH	676.	2513	14,159	2583	11,573	42
16	EPCC	494.	2513	14,159	2583	11,573	42
16	SHMEM	471.	2513	14,160	2581	11,576	42
32	MPICH	1367.	2536	20,153	2696	17,454	43
32	EPCC	737.	2536	20,153	2696	17,454	43
32	SHMEM	695.	2536	20,121	2694	17,424	43

Table 1: Parallel CVODE test results vs problem size and message-passing library

but EPCC is a very close second, and MPICH loses considerable efficiency by comparison, as the problem size grows. This means that the highly portable MPI version of CVODE, with an appropriate choice of MPI implementation, is fully competitive with the Cray-specific version using the SHMEM library. While the overall costs do not prepresent a well-scaled parallel algorithm (because of the preconditioner choice), the cost per function evaluation is quite flat for EPCC and SHMEM, at .033 to .037 (for MPICH it ranges from .044 to .068).

For tests that demonstrate speedup from parallelism, we consider runs with fixed problem size: `MX` = 800, `MY` = 400. Here we also fix the vertical subgrid dimension at `MYSUB` = 50 and the vertical processor array dimension at `NPEY` = 8, but vary the corresponding horizontal sizes. We take `NPEX` = 8, 16, and 32, with `MXSUB` = 100, 50, and 25, respectively. The runs for the three cases and three message-passing libraries all show very good agreement in solution values and performance counts. The run times for EPCC are 947, 494, and 278, showing speedups of 1.92 and 1.78 as the number of processors is doubled (twice). For the SHMEM runs, the times were slightly lower and the ratios were 1.98 and 1.91. For MPICH, consistent with the earlier runs, the run times were considerably higher, and in fact show speedup ratios of only 1.54 and 1.03.

References

[1] R Falgout and UM Yang. Hypre user's manual. Technical report, LLNL, 2015.

[2] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v3.2.1. Technical Report UCRL-SM-208108, LLNL, 2018.

[3] M. R. Wittman. Testing of PVODE, a Parallel ODE Solver. Technical Report UCRL-ID-125562, LLNL, August 1996.