Example Programs for cvode v5.2.0

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

March 31, 2020
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.
# Contents

1 Introduction ........................................................................................................... 1

2 Serial example problems ...................................................................................... 5
   2.1 A dense example: cvRoberts_dns .................................................................. 5
   2.2 A banded example: cvAdvDiff_bnd ............................................................. 7
   2.3 A Krylov example: cvDiurnal_kry ............................................................. 10

3 Parallel example problems .................................................................................... 14
   3.1 A nonstiff example: cvAdvDiff_non_p ......................................................... 14
   3.2 A user preconditioner example: cvDiurnal_kry_p ....................................... 16
   3.3 A CVBBDPRE preconditioner example: cvDiurnal_kry_bbd_p .................... 18

4 hypre example problems ......................................................................................... 21
   4.1 A nonstiff example: cvAdvDiff_non_ph ...................................................... 21

5 CUDA example problems ....................................................................................... 22
   5.1 An unpreconditioned Krylov example: cvAdvDiff_kry_cuda ...................... 22

6 RAJA example problems ....................................................................................... 23
   6.1 An unpreconditioned Krylov example: cvAdvDiff_kry_raja ....................... 23

7 Fortran example problems ..................................................................................... 24
   7.1 A serial example: fcvDiurnal_kry ............................................................... 24
   7.2 A parallel example: fcvDiag_kry_bbd_p .................................................... 25

8 Parallel tests ........................................................................................................... 28

References .................................................................................................................. 30
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 fixed-point 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 NVVECTOR_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 NVVECTOR_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, CVLS 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, CVLS 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 cvls 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 fixed-point 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 cvls 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 preconditioner 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 NVVECTOR_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 fixed-point 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/FIXEDPOINT).

• `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 linear solver interface cvls 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:

\[ \begin{align*}
\dot{y}_1 &= -0.04 \cdot y_1 + 10^4 \cdot y_2 \cdot y_3 \\
\dot{y}_2 &= 0.04 \cdot y_1 - 10^4 \cdot y_2 \cdot y_3 - 3 \cdot 10^7 \cdot y_2^2 \\
\dot{y}_3 &= 3 \cdot 10^7 \cdot y_2^2
\end{align*} \]  

(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 sunlinsol_dense.h file is the header file for the dense implementation 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 NVector 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 §6.3. The macro `SM_ELEMENT_D` is fully described in §7.3.

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 dependent 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 §4.5.5 for a detailed description of this call.

The call to SUNDenseMatrix (see §7.3) creates a \( \text{NEQ} \times \text{NEQ} \) dense SUNMATRIX object to use within the Newton solve in CVODE. The following call to SUNLinSol_Dense (see §8.5) creates the dense SUNLINSOL object that will perform the linear solves within the Newton method. These are attached to the CVLS linear solver interface with the call to CVodeSetLinearSolver (see §4.5.3), and the subsequent call to CVodeSetJacFn (see §4.5.7) 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 \( t = \text{tout} \), putting the computed value of \( y(\text{tout}) \) into y, with \( t = \text{tout} \). 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 §4.5.6 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*** functions to obtain the relevant counters, and then prints them. Specifically, these are: the
cumulative number of steps (\(\text{nst}\)), the number of \(f\) evaluations (\(\text{nfe}\)) (excluding those for difference-quotient Jacobian evaluations), the number of matrix factorizations (\(\text{nsetups}\)), the number of \(f\) evaluations for Jacobian evaluations (\(\text{nfeLS} = 0\) here), the number of Jacobian evaluations (\(\text{nje}\)), the number of nonlinear (Newton) iterations (\(\text{nni}\)), the number of nonlinear convergence failures (\(\text{ncfn}\)), the number of local error test failures (\(\text{netf}\)), and the number of \(g\) (root function) evaluations (\(\text{nge}\)). These optional outputs are described in §4.5.9.

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 \text{sunmatrix}. Here the problem size is small, so we need not worry about the inefficiency of using \text{NV_Ith_S} and \text{SM_ELEMENT_D} to access \text{N_Vector} and \text{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 \text{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 | \(y = 9.055097e-01\) | \(2.240338e-05\) | \(9.44793e-02\) |
| 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 | \(y = 1.832299e-01\) | \(8.944378e-07\) | \(8.16792e-01\) |
| At t = 4.0000e+04 | \(y = 3.898902e-02\) | \(1.622006e-07\) | \(9.610108e-02\) |
| At t = 4.0000e+05 | \(y = 4.936383e-03\) | \(1.984224e-08\) | \(9.950636e-01\) |
| At t = 4.0000e+06 | \(y = 5.168093e-04\) | \(2.068293e-09\) | \(9.94832e-01\) |
| At t = 2.0790e+07 | \(y = 1.000000e-04\) | \(4.000397e-10\) | \(9.99000e-01\) |

rootsfound[] = -1 0

At t = 4.0000e+07 | \(y = 5.202440e-05\) | \(2.081083e-10\) | \(9.994800e-01\)
At t = 4.0000e+08 | \(y = 5.201061e-06\) | \(2.080435e-11\) | \(9.99998e-01\)
At t = 4.0000e+09 | \(y = 5.258603e-07\) | \(2.103442e-12\) | \(9.999999e-01\)
At t = 4.0000e+10 | \(y = 6.934511e-08\) | \(2.773804e-13\) | \(9.999999e-01\)

Final Statistics:
\(\text{nst} = 542\) \(\text{nfe} = 754\) \(\text{nsetups} = 107\) \(\text{nfeLS} = 0\) \(\text{nni} = 751\) \(\text{ncfn} = 0\) \(\text{netf} = 22\) \(\text{nge} = 570\)

2.2 A banded example: cvAdvDiff_bnd

The example program \text{cvAdvDiff_bnd.c} solves the semi-discretized form of the 2-D advection-diffusion equation
\[
\frac{\partial v}{\partial t} = \partial^2 v / \partial x^2 + 0.5 \partial^2 v / \partial x \partial y + \partial^2 v / \partial y^2
\]
on a rectangle, with zero Dirichlet boundary conditions. The PDE is discretized with standard central finite differences on a $(MX+2) \times (MY+2)$ mesh, giving an ODE system of size $MX \times MY$. The discrete value $v_{ij}$ approximates $v$ at $x = i \Delta x$, $y = j \Delta y$. The ODEs are

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

where $1 \leq i \leq MX$ and $1 \leq j \leq MY$. 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 $u_{(j-1)(i-1)+MY} = v_{ij}$, 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 and sunlinsol_band.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 §7.4). 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 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 CVodeSSetTolerances indicates scalar relative and absolute tolerances, and values reltol and abstol are passed. The call to SUNBandMatrix (see §7.4) creates a banded SUNMATRIX Jacobian template, and specifies that both half-bandwidths of the Jacobian are equal to $MY$. The calls to SUNBandLinearSolver (see §8.6) and CVodeSetLinearSolver (see §4.5.3) specifies the SUNLINSOL_BAND linear solver to the cvls interface. The call to CVodeSetJacFn (see §4.5.7) 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 Jacobian 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 §4.5.9. 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
\[
\frac{\partial f_{ij}}{\partial v_{ij}} = -2[(\Delta x)^{-2} + (\Delta y)^{-2}]
\]
\[
\frac{\partial f_{ij}}{\partial v_{i\pm1,j}} = (\Delta x)^{-2} \pm .5(2\Delta x)^{-1}, \quad \frac{\partial f_{ij}}{\partial v_{i,j\pm1}} = (\Delta y)^{-2}.
\]

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 $k = j-1 + (i-1)\times MY$ 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 $v_{ij}$ 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 to 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.

---

<table>
<thead>
<tr>
<th>cvAdvDiff_bnd sample output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D Advection-Diffusion Equation</td>
</tr>
<tr>
<td>Mesh dimensions = 10 X 5</td>
</tr>
<tr>
<td>Total system size = 50</td>
</tr>
<tr>
<td>Tolerance parameters: reltol = 0 abstol = 1e-05</td>
</tr>
<tr>
<td>At t = 0.0  max.norm(u) =  8.954716e+01</td>
</tr>
<tr>
<td>At t = 0.10 max.norm(u) =  4.132889e+00  nst =  85</td>
</tr>
<tr>
<td>At t = 0.20 max.norm(u) =  1.039294e+00  nst =  103</td>
</tr>
<tr>
<td>At t = 0.30 max.norm(u) =  2.979829e-01  nst =  113</td>
</tr>
<tr>
<td>At t = 0.40 max.norm(u) =  8.765774e-02  nst =  120</td>
</tr>
<tr>
<td>At t = 0.50 max.norm(u) =  2.625637e-02  nst =  126</td>
</tr>
<tr>
<td>At t = 0.60 max.norm(u) =  7.830425e-03  nst =  130</td>
</tr>
<tr>
<td>At t = 0.70 max.norm(u) =  2.329387e-03  nst =  134</td>
</tr>
<tr>
<td>At t = 0.80 max.norm(u) =  6.953434e-04  nst =  137</td>
</tr>
<tr>
<td>At t = 0.90 max.norm(u) =  2.115983e-04  nst =  140</td>
</tr>
</tbody>
</table>
At \( t = 1.00 \) \ max. norm \((u) = 6.556853 \times 10^{-5} \) \ nst = 142

Final Statistics:

\[ \text{nst} = 142 \quad \text{nfe} = 173 \quad \text{nsetups} = 23 \quad \text{nfeLS} = 0 \quad \text{nje} = 3 \]
\[ \text{nni} = 170 \quad \text{ncfn} = 0 \quad \text{netf} = 3 \]

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 CVLS 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 non-linear diurnal kinetics, horizontal advection and diffusion, and nonuniform vertical diffusion. The PDEs can be written as

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

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

\[
R^1(c^1, c^2, t) = -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t)c^3 + q_4(t)c^2, \\
R^2(c^1, c^2, t) = q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t)c^2.
\]

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

\[
q_i(t) = \begin{cases} 
\exp[-a_i / \sin \omega t], & \text{for } \sin \omega t > 0 \\
0, & \text{for } \sin \omega t \leq 0
\end{cases} \quad (i = 3, 4),
\]

where \( \omega = \pi/43200, \quad a_3 = 22.62, \quad a_4 = 7.601. \) 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

\[
c^1(x, y, 0) = 10^6 \alpha(x) \beta(y), \quad c^2(x, y, 0) = 10^{12} \alpha(x) \beta(y), \\
\alpha(x) = 1 - (0.1x - 1)^2 + (0.1x - 1)^4/2, \\
\beta(y) = 1 - (0.1y - 4)^2 + (0.1y - 4)^4/2.
\]

For this example, the equations (4) are discretized spatially with standard central finite differences on a \( 10 \times 10 \) 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 SUNLinSol_SPGMR. 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 \texttt{SUNLinSol\_SPGMR} to create the \texttt{SPGMR} linear solver with left preconditioning, and the default value (indicated by a zero argument) for \texttt{maxl}. It then calls \texttt{CVodeSetLinearSolver} (see §4.5.3) to attach this linear solver to the \texttt{cvls} interface. The call to \texttt{CVodeSetJacTimes} specifies a user-supplied function for Jacobian-vector products (the \texttt{NULL} argument specifies that no Jacobian-vector setup routine is needed). Next, user-supplied preconditioner setup and solve functions, \texttt{Precond} and \texttt{PSolve}, are specified. See §4.5.7 for details on the \texttt{CVodeSetPreconditioner} function.

For a sequence of \texttt{tout} values, \texttt{CVode} is called in the \texttt{CV\_NORMAL} mode, sampled output is printed, and the return value is tested for error conditions. After that, \texttt{PrintFinalStats} is called to get and print final statistics, and memory is freed by calls to \texttt{N\_VDestroy}, \texttt{FreeUserData}, and \texttt{CVodeFree}. The printed statistics include various counters, such as the total numbers of steps (\texttt{nst}), of \texttt{f} evaluations (excluding those for \texttt{Jv} product evaluations) (\texttt{nfe}), of \texttt{f} evaluations for \texttt{Jv} evaluations (\texttt{nfeLS}), of nonlinear iterations (\texttt{nni}), of linear (Krylov) iterations (\texttt{nl}), of preconditioner setups (\texttt{nsetups}), of preconditioner evaluations (\texttt{npe}), and of preconditioner solves (\texttt{nps}), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator \texttt{CVode}, denoted \texttt{lenr} and \texttt{leni}, and those used by \texttt{cvls}, denoted \texttt{lenrLS} and \texttt{leniLS}. All of these optional outputs are described in §4.5.9. The \texttt{PrintFinalStats} function is suitable for general use in applications of \texttt{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 \texttt{NVECTOR\_SERIAL}, a vector of type \texttt{N\_Vector} works with a one-dimensional contiguous array of data components. The macro \texttt{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 \texttt{NV\_DATA\_S} macro for efficient \texttt{N\_Vector} access. The \texttt{NV\_DATA\_S} macro gives a pointer to the first component of an \texttt{N\_Vector} which we pass to the \texttt{IJKth} macro to do an \texttt{N\_Vector} access.

The preconditioner used here is the block-diagonal part of the true Newton matrix. It is generated and factored in the \texttt{Precond} routine (see §4.6.10) and backsolved in the \texttt{PSolve} routine (see §4.6.9). Its diagonal blocks are \( 2 \times 2 \) matrices that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian, \( J_{bd} \), is saved in separate storage each time it is generated, on calls to \texttt{Precond} with \texttt{jok == SUNFALSE}. On calls with \texttt{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 \texttt{Precond} with \texttt{jok == SUNTRUE} can only occur after a prior call with \texttt{jok == SUNFALSE}.) The \texttt{Precond} routine must also set the value of \texttt{jcur}, i.e. \texttt{*jcurPtr}, to \texttt{SUNTRUE} when \( J_{bd} \) is re-evaluated, and \texttt{SUNFALSE} otherwise, to inform \texttt{cvls} 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 \texttt{SUNLINSOL\_DENSE} linear solver module serves as the interface to dense matrix solvers for the main \texttt{SUNDIALS} solvers, the underlying algebraic operations operate on dense matrices with \texttt{realtype **} as the underlying dense matrix type. To avoid the extra layer of function calls and dense matrix and linear solver data structures, \texttt{cvDiurnal\_kry.c} uses underlying small dense functions for all operations on the \( 2 \times 2 \) preconditioner blocks. Thus it includes \texttt{sundials\_dense.h}, and calls the small dense matrix functions \texttt{newDenseMat}, \texttt{newIndexArray}, \texttt{denseCopy}, \texttt{denseScale}, \texttt{denseAddIdentity}, \texttt{denseGETRF}, and \texttt{denseGETRS}. The macro \texttt{IJKth} 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 retrieve 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.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$n_{steps}$</th>
<th>order</th>
<th>stepsize</th>
<th>$c_1$ (bot.left/middle/top rt.)</th>
<th>$c_2$ (bot.left/middle/top rt.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.20e+03</td>
<td>219</td>
<td>5</td>
<td>1.59e+02</td>
<td>1.047e+04, 2.964e+04, 1.119e+04</td>
<td>2.527e+11, 7.154e+11, 2.700e+11</td>
</tr>
<tr>
<td>1.44e+04</td>
<td>251</td>
<td>5</td>
<td>3.77e+02</td>
<td>6.659e+06, 5.316e+06, 7.301e+06</td>
<td>2.582e+11, 2.057e+11, 2.833e+11</td>
</tr>
<tr>
<td>2.16e+04</td>
<td>277</td>
<td>5</td>
<td>2.75e+02</td>
<td>2.665e+07, 1.036e+07, 2.931e+07</td>
<td>2.993e+11, 1.028e+11, 3.313e+11</td>
</tr>
<tr>
<td>2.88e+04</td>
<td>307</td>
<td>4</td>
<td>2.03e+02</td>
<td>8.702e+06, 1.292e+07, 9.650e+06</td>
<td>3.380e+11, 5.029e+11, 3.751e+11</td>
</tr>
<tr>
<td>3.60e+04</td>
<td>338</td>
<td>5</td>
<td>9.92e+01</td>
<td>1.404e+04, 2.029e+04, 1.561e+04</td>
<td>3.387e+11, 4.894e+11, 3.765e+11</td>
</tr>
<tr>
<td>4.32e+04</td>
<td>393</td>
<td>4</td>
<td>2.57e+02</td>
<td>-7.891e-06, -1.123e-06, -8.723e-06</td>
<td>3.382e+11, 1.355e+11, 3.804e+11</td>
</tr>
<tr>
<td>5.04e+04</td>
<td>421</td>
<td>5</td>
<td>4.96e+02</td>
<td>-1.595e-08, -1.771e-06, -3.117e-08</td>
<td>3.358e+11, 4.930e+11, 3.864e+11</td>
</tr>
<tr>
<td>5.76e+04</td>
<td>439</td>
<td>4</td>
<td>1.19e+02</td>
<td>-2.127e-06, -1.513e-04, -2.951e-06</td>
<td>3.320e+11, 9.650e+11, 3.909e+11</td>
</tr>
<tr>
<td>6.48e+04</td>
<td>458</td>
<td>5</td>
<td>6.60e+02</td>
<td>-1.084e-09, -7.686e-08, -1.499e-09</td>
<td>3.313e+11, 8.922e+11, 3.963e+11</td>
</tr>
<tr>
<td>7.20e+04</td>
<td>469</td>
<td>5</td>
<td>6.60e+02</td>
<td>-1.093e-10, -7.736e-09, -1.510e-10</td>
<td>3.330e+11, 6.186e+11, 4.039e+11</td>
</tr>
<tr>
<td>7.92e+04</td>
<td>480</td>
<td>5</td>
<td>6.60e+02</td>
<td>-1.093e-10, -7.736e-09, -1.510e-10</td>
<td>3.330e+11, 6.186e+11, 4.039e+11</td>
</tr>
<tr>
<td>c1 (bot. left/middle/top rt.)</td>
<td>-1.624e-12</td>
<td>-1.151e-10</td>
<td>-2.246e-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2 (bot. left/middle/top rt.)</td>
<td>3.334e+11</td>
<td>6.669e+11</td>
<td>4.120e+11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 8.64e+04</td>
<td>no. steps = 491</td>
<td>order = 5</td>
<td>stepsize = 6.60e+02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1 (bot. left/middle/top rt.)</td>
<td>3.501e-14</td>
<td>2.474e-12</td>
<td>4.816e-14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2 (bot. left/middle/top rt.)</td>
<td>3.352e+11</td>
<td>9.107e+11</td>
<td>4.163e+11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Final Statistics..**

| lenrw = 2689 | leniw = 53 |
| lenrWS = 2454 | leniWS = 42 |
| nst = 491 |
| nfe = 630 | nfeLS = 0 |
| nni = 627 | nli = 643 |
| nsetups = 82 | netf = 29 |
| npe = 9 | nps = 1214 |
| ncfn = 0 | nclf = 0 |
3 Parallel example problems

3.1 A nonstiff example: cvAdvDiff_non_p

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

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0.5 \frac{\partial u}{\partial x}
\]

(7)

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

\[
\begin{align*}
    u(t, 0) &= 0 , & u(t, 2) &= 0 , \\
    u(0, x) &= x(2 - x)e^{2x} .
\end{align*}
\]

(8)

A system of $M_X$ ODEs is obtained by discretizing the $x$-axis with $M_X + 2$ 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 $\Delta x = 2/(M_X + 1)$, the resulting system of ODEs, $\dot{u} = f(t, u)$, can now be written:

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

(9)

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

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 fixed-point nonlinear solver. 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 (\texttt{my\_base}+1 to \texttt{my\_base+my\_length}) corresponding to the local part of the solution vector \texttt{u}, and then to load the corresponding initial values. The PrintFinalStats routine uses \texttt{CVodeGet***} calls to get various counters, and then prints these. The counters are: \texttt{nst} (number of steps), \texttt{nfe} (number of \texttt{f} evaluations), \texttt{nni} (number of nonlinear iterations), \texttt{netf} (number of error test failures), and \texttt{ncfn} (number of nonlinear convergence failures). This routine is suitable for general use with \texttt{CVODE} applications to nonstiff problems.

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

The \texttt{cvAdvDiff\_non\_p.c} file includes a routine \texttt{check\_flag} that checks the return values from calls in \texttt{main}. This routine was written to be used by any parallel \texttt{SUNDIALS} application.

The output below is for \texttt{cvAdvDiff\_non\_p} with \texttt{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 \texttt{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

<table>
<thead>
<tr>
<th>Time (t)</th>
<th>Max. Norm. (u)</th>
<th>Steps (nst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 0.00</td>
<td>1.569909e+01</td>
<td>0</td>
</tr>
<tr>
<td>t = 0.50</td>
<td>3.052881e+00</td>
<td>113</td>
</tr>
<tr>
<td>t = 1.00</td>
<td>8.753188e-01</td>
<td>191</td>
</tr>
<tr>
<td>t = 1.50</td>
<td>2.494926e-01</td>
<td>265</td>
</tr>
<tr>
<td>t = 2.00</td>
<td>7.109437e-02</td>
<td>332</td>
</tr>
<tr>
<td>t = 2.50</td>
<td>2.026176e-02</td>
<td>405</td>
</tr>
<tr>
<td>t = 3.00</td>
<td>5.769193e-03</td>
<td>468</td>
</tr>
<tr>
<td>t = 3.50</td>
<td>1.651582e-03</td>
<td>541</td>
</tr>
<tr>
<td>t = 4.00</td>
<td>4.764197e-04</td>
<td>623</td>
</tr>
<tr>
<td>t = 4.50</td>
<td>1.370219e-04</td>
<td>704</td>
</tr>
<tr>
<td>t = 5.00</td>
<td>3.990405e-05</td>
<td>785</td>
</tr>
</tbody>
</table>

Final Statistics:

\texttt{nst} = 785, \texttt{nfe} = 1441, \texttt{nni} = 1438, \texttt{ncfn} = 140, \texttt{netf} = 5
3.2 A user preconditioner example: cvDiurnal_kry_p

As an example of using CVODE with the Krylov linear solver SUNLINSOL, SPGMR, CVLS linear solver interface, 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 \( M_{\text{SUB}} \times M_{\text{SUB}} \) of the \( x \)-\( y \) grid. If there are \( N_{\text{PEX}} \) processors in the \( x \) direction and \( N_{\text{PEY}} \) processors in the \( y \) direction, then the overall grid size is \( M \times M \) with \( M = N_{\text{PEX}} \times M_{\text{SUB}} \) and \( M = N_{\text{PEY}} \times M_{\text{SUB}} \), and the size of the ODE system is \( 2 \times M \times M \).

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 \( 2 \times 2 \) 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 \( 2 \times 2 \) array) with a \( 5 \times 5 \) 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

\[
t = 7.20e+03 \quad \text{no. steps} = 219 \quad \text{order} = 5 \quad \text{stepsize} = 1.59e+02
\]

At bottom left: \( c_1, c_2 = 1.047e+04, 2.527e+11 \)
At top right: \( c_1, c_2 = 1.119e+04, 2.700e+11 \)
<table>
<thead>
<tr>
<th>t</th>
<th>no. steps</th>
<th>order</th>
<th>stepsize</th>
<th>At bottom left: c1, c2</th>
<th>At top right: c1, c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.44e+04</td>
<td>251</td>
<td>5</td>
<td>3.77e+02</td>
<td>6.659e+06, 2.582e+11</td>
<td>7.301e+06, 2.833e+11</td>
</tr>
<tr>
<td>2.16e+04</td>
<td>277</td>
<td>5</td>
<td>2.75e+02</td>
<td>2.665e+07, 2.993e+11</td>
<td>2.931e+07, 3.313e+11</td>
</tr>
<tr>
<td>2.88e+04</td>
<td>321</td>
<td>3</td>
<td>5.19e+01</td>
<td>8.702e+06, 3.380e+11</td>
<td>9.650e+06, 3.751e+11</td>
</tr>
<tr>
<td>3.60e+04</td>
<td>384</td>
<td>4</td>
<td>8.70e+01</td>
<td>1.404e+04, 3.387e+11</td>
<td>1.561e+04, 3.765e+11</td>
</tr>
<tr>
<td>4.32e+04</td>
<td>456</td>
<td>4</td>
<td>8.77e+02</td>
<td>1.077e-06, 3.804e+11</td>
<td>4.057e-07, 3.804e+11</td>
</tr>
<tr>
<td>5.04e+04</td>
<td>471</td>
<td>4</td>
<td>3.29e+02</td>
<td>-1.176e-08, 3.538e+11</td>
<td>-5.053e-08, 3.864e+11</td>
</tr>
<tr>
<td>5.76e+04</td>
<td>488</td>
<td>5</td>
<td>3.95e+02</td>
<td>-9.464e-11, 3.320e+11</td>
<td>-3.463e-10, 3.909e+11</td>
</tr>
<tr>
<td>6.48e+04</td>
<td>501</td>
<td>5</td>
<td>6.20e+02</td>
<td>5.057e-11, 3.313e+11</td>
<td>1.868e-10, 3.963e+11</td>
</tr>
<tr>
<td>7.20e+04</td>
<td>512</td>
<td>5</td>
<td>6.20e+02</td>
<td>-4.454e-11, 3.330e+11</td>
<td>-1.629e-10, 4.039e+11</td>
</tr>
<tr>
<td>7.92e+04</td>
<td>524</td>
<td>5</td>
<td>6.20e+02</td>
<td>-2.189e-13, 3.334e+11</td>
<td>-8.112e-13, 4.120e+11</td>
</tr>
<tr>
<td>8.64e+04</td>
<td>535</td>
<td>5</td>
<td>6.20e+02</td>
<td>1.080e-15, 3.352e+11</td>
<td>3.729e-15, 4.163e+11</td>
</tr>
</tbody>
</table>

Final Statistics:

lenrw = 2689  leniw = 144
lenrwls = 2454  leniwls = 126
nst = 535
nfe = 687  nfe = 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 CVBBDBPRE memory (CVBBDPrecReInit), as well as the preconditioner type (SUNLinSol_SPGMRSetPrecType).

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.

<table>
<thead>
<tr>
<th>t</th>
<th>no. steps</th>
<th>order</th>
<th>stepsize</th>
<th>At bottom left: c1, c2</th>
<th>At top right: c1, c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.20e+03</td>
<td>190</td>
<td>5</td>
<td>1.61e+02</td>
<td>1.047e+04, 2.527e+11</td>
<td>1.119e+04, 2.700e+11</td>
</tr>
<tr>
<td>1.44e+04</td>
<td>221</td>
<td>5</td>
<td>3.85e+02</td>
<td>6.659e+06, 2.582e+11</td>
<td>7.301e+06, 2.833e+11</td>
</tr>
<tr>
<td>2.16e+04</td>
<td>247</td>
<td>5</td>
<td>3.00e+02</td>
<td>2.665e+07, 2.993e+11</td>
<td>2.931e+07, 3.313e+11</td>
</tr>
<tr>
<td>2.88e+04</td>
<td>272</td>
<td>5</td>
<td>2.13e+02</td>
<td>8.702e+06, 3.380e+11</td>
<td>9.650e+06, 3.751e+11</td>
</tr>
<tr>
<td>3.60e+04</td>
<td>311</td>
<td>4</td>
<td>9.70e+01</td>
<td>8.702e+06, 3.380e+11</td>
<td>9.650e+06, 3.751e+11</td>
</tr>
</tbody>
</table>

---

--- cvDiurnal_kry_bbd_p output ---
At bottom left:  \( c_1, c_2 = 1.404 e+04 \quad 3.387 e+11 \)
At top right:  \( c_1, c_2 = 1.561 e+04 \quad 3.765 e+11 \)

\( t = 4.32 e+04 \quad \text{no. steps} = 368 \quad \text{order} = 4 \quad \text{stepsize} = 3.95 e+02 \)
At bottom left:  \( c_1, c_2 = 6.243 e-08 \quad 3.382 e+11 \)
At top right:  \( c_1, c_2 = 7.021 e-08 \quad 3.804 e+11 \)

\( t = 5.04 e+04 \quad \text{no. steps} = 388 \quad \text{order} = 5 \quad \text{stepsize} = 4.74 e+02 \)
At bottom left:  \( c_1, c_2 = -1.096 e-10 \quad 3.320 e+11 \)
At top right:  \( c_1, c_2 = 7.021 e-08 \quad 3.804 e+11 \)

Final Statistics:

\begin{align*}
\text{lenrw} &= 2689 \quad \text{leniw} = 144 \\
\text{lenrwlsl} &= 2454 \quad \text{leniwsl} = 126 \\
\text{nsetups} &= 76 \quad \text{nfels} = 509 \\
\text{nfe} &= 580 \quad \text{nni} = 577 \quad \text{nli} = 509 \\
\text{nfels} &= 509 \quad \text{npe} = 8 \quad \text{nps} = 1029 \\
\text{ncfn} &= 0 \quad \text{ncf1} = 0 \\
\end{align*}

In CVBBDPRE: real/integer local work space sizes = 1300, 192

no. local evals. = 176

Preconditioner type is: \( \text{jpre} = \text{PREC_RIGHT} \)
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 = 406  order = 5  stepsize = 6.68e+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 = 421  order = 4  stepsize = 1.46e+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 = 448  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 = 6.48e+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 = 7.20e+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 = 2689  leniw = 144
lenrwls = 2454  leniwls = 126
nst = 472
nfe = 603  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 \texttt{hypre} example problems

4.1 A nonstiff example: \texttt{cvAdvDiff\_non\_ph}

This example is same as \texttt{cvAdvDiff\_non\_p}, except that it uses the \texttt{hypre} vector type instead of the \texttt{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 \texttt{hypre} library [1] is helpful.

We use the \texttt{hypre} IJ vector interface to allocate the template vector and create parallel partitioning:

\begin{verbatim}
HYPRE_IJVectorCreate(comm, my_base, my_base + local_N - 1, &Uij);
HYPRE_IJVectorSetObjectType(Uij, HYPRE_PARCSR);
HYPRE_IJVectorInitialize(Uij);
\end{verbatim}

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 \texttt{SetIC(...)\hspace{1em}} function. We complete the \texttt{hypre} vector assembly by:

\begin{verbatim}
HYPRE_IJVectorAssemble(Uij);
HYPRE_IJVectorGetObject(Uij, (void**) &Upar);
\end{verbatim}

The assemble call is collective and it makes \texttt{hypre} vector ready to use. This sets the handle \texttt{Upar} to the actual \texttt{hypre} vector. The handle is then passed to the \texttt{N\_VMake} function, which creates the template \texttt{N\_Vector} as a wrapper around the \texttt{hypre} vector. All other vectors in the computation are created by cloning the template vector. The template vector does not own the underlying \texttt{hypre} vector, and it is the user’s responsibility to destroy it using a \texttt{HYPRE\_IJVectorDestroy(Uij)} call after the template vector has been destroyed. This function will destroy both the \texttt{hypre} vector and its IJ interface.

To access individual elements of solution vectors \( u \) and \( u\dot{} \) in the residual function, the user needs to extract the \texttt{hypre} vector first by calling \texttt{N\_VGetVector\_ParHyp}, and then use \texttt{hypre} methods from that point on.

Notes

- At this point interfaces to \texttt{hypre} solvers and preconditioners are not available. They will be provided in subsequent \texttt{sundials} releases. The interface to \texttt{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 a 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.h provides the prototypes for CVodeSetLinearSolver, which sets the iterative linear solver for CVODE, and CVodeSetJacTimes, which sets the pointer to the 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 their 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.

<table>
<thead>
<tr>
<th>2-D Advection-Diffusion Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh dimensions = 10 X 5</td>
</tr>
<tr>
<td>Total system size = 50</td>
</tr>
<tr>
<td>Tolerance parameters: reltol = 0</td>
</tr>
<tr>
<td>abstol = 1e-05</td>
</tr>
<tr>
<td>At t = 0   max. norm(u) = 8.954716e+01</td>
</tr>
<tr>
<td>At t = 0.10 max. norm(u) = 4.132884e+00 nst = 82</td>
</tr>
<tr>
<td>At t = 0.20 max. norm(u) = 1.039293e+00 nst = 102</td>
</tr>
<tr>
<td>At t = 0.30 max. norm(u) = 2.979816e-01 nst = 112</td>
</tr>
<tr>
<td>At t = 0.40 max. norm(u) = 8.765538e-02 nst = 119</td>
</tr>
<tr>
<td>At t = 0.50 max. norm(u) = 2.625408e-02 nst = 125</td>
</tr>
<tr>
<td>At t = 0.60 max. norm(u) = 7.826077e-03 nst = 129</td>
</tr>
<tr>
<td>At t = 0.70 max. norm(u) = 2.326537e-03 nst = 133</td>
</tr>
<tr>
<td>At t = 0.80 max. norm(u) = 6.891180e-04 nst = 137</td>
</tr>
<tr>
<td>At t = 0.90 max. norm(u) = 2.039853e-04 nst = 140</td>
</tr>
<tr>
<td>At t = 1.00 max. norm(u) = 5.925586e-05 nst = 143</td>
</tr>
</tbody>
</table>

Final Statistics..

| lenrv  | 739 |
| leniw  | 66  |
| lenrvLS| 654 |
| leniwLS| 54  |
| nst    | 143 |
| nfe    | 206 |
| nfeLS  | 0   |
| nni    | 203 |
| nli    | 225 |
| nsetups| 0   |
| netf   | 2   |
| npe    | 0   |
| nps    | 0   |
| ncfn   | 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.

<table>
<thead>
<tr>
<th>Time (t)</th>
<th>Max. Norm (u)</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.954716e+01</td>
<td>82</td>
</tr>
<tr>
<td>0.10</td>
<td>4.132884e+00</td>
<td>102</td>
</tr>
<tr>
<td>0.20</td>
<td>1.039293e+00</td>
<td>112</td>
</tr>
<tr>
<td>0.30</td>
<td>2.979816e-01</td>
<td>119</td>
</tr>
<tr>
<td>0.40</td>
<td>8.765538e-02</td>
<td>119</td>
</tr>
<tr>
<td>0.50</td>
<td>2.625408e-02</td>
<td>125</td>
</tr>
<tr>
<td>0.60</td>
<td>7.826077e-03</td>
<td>129</td>
</tr>
<tr>
<td>0.70</td>
<td>2.326537e-03</td>
<td>133</td>
</tr>
<tr>
<td>0.80</td>
<td>6.891180e-04</td>
<td>137</td>
</tr>
<tr>
<td>0.90</td>
<td>2.039853e-04</td>
<td>140</td>
</tr>
<tr>
<td>1.00</td>
<td>5.925586e-05</td>
<td>143</td>
</tr>
</tbody>
</table>

Final Statistics...

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lenrw</td>
<td>739</td>
</tr>
<tr>
<td>leniw</td>
<td>66</td>
</tr>
<tr>
<td>lenrwLS</td>
<td>654</td>
</tr>
<tr>
<td>leniwLS</td>
<td>54</td>
</tr>
<tr>
<td>nst</td>
<td>143</td>
</tr>
<tr>
<td>nfe</td>
<td>206</td>
</tr>
<tr>
<td>nfeLS</td>
<td>0</td>
</tr>
<tr>
<td>nni</td>
<td>203</td>
</tr>
<tr>
<td>nli</td>
<td>225</td>
</tr>
<tr>
<td>nsetups</td>
<td>0</td>
</tr>
<tr>
<td>netf</td>
<td>2</td>
</tr>
<tr>
<td>npe</td>
<td>0</td>
</tr>
<tr>
<td>nps</td>
<td>0</td>
</tr>
<tr>
<td>ncfn</td>
<td>0</td>
</tr>
</tbody>
</table>
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 §5.2.2 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 FNINITS, FSUNSPPGMRINIT, FSUNSPPGMRSETGSTYPE, FCVMALLOC, FCVLSETINT, and FCVLSETPREC, to initialize the NVECTOR_SERIAL module, the SUNLINSOL_SPGMR module, the main solver memory, and the CVLS 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, Jbd, 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 Jbd 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.

---
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 0.715376E+12 0.269977E+12

t = 0.216E+05  nst = 277  q = 5  h = 0.274584E+03
  c1 (bot.left/middle/top rt.) = 0.870209E+07 0.129197E+08 0.965002E+07
  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
---

24
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 \(N = 10\times\text{NPES}\), where \text{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 `FCVLSINIT` 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 = 213
t = 0.20E+00 no. steps = 222 no. f-s = 262
t = 0.30E+00 no. steps = 247 no. f-s = 288
t = 0.40E+00 no. steps = 265 no. f-s = 307
t = 0.50E+00 no. steps = 278 no. f-s = 321
t = 0.60E+00 no. steps = 290 no. f-s = 333
t = 0.70E+00 no. steps = 300 no. f-s = 344
t = 0.80E+00 no. steps = 307 no. f-s = 351
t = 0.90E+00 no. steps = 312 no. f-s = 357
t = 1.00E+01 no. steps = 317 no. f-s = 362

Max. absolute error is 0.90E-08

Final statistics:

number of steps = 317 number of f evals. = 362
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 = 349 92
linear solver real/int workspace sizes = 294 78

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

Preconditioning on right

\[
\begin{align*}
\text{t} &= 0.10 \times 10^0 & \text{no. steps} &= 174 & \text{no. f-s} &= 213 \\
\text{t} &= 0.20 \times 10^0 & \text{no. steps} &= 222 & \text{no. f-s} &= 262 \\
\text{t} &= 0.30 \times 10^0 & \text{no. steps} &= 247 & \text{no. f-s} &= 288 \\
\text{t} &= 0.40 \times 10^0 & \text{no. steps} &= 265 & \text{no. f-s} &= 307 \\
\text{t} &= 0.50 \times 10^0 & \text{no. steps} &= 278 & \text{no. f-s} &= 321 \\
\text{t} &= 0.60 \times 10^0 & \text{no. steps} &= 290 & \text{no. f-s} &= 333 \\
\text{t} &= 0.70 \times 10^0 & \text{no. steps} &= 300 & \text{no. f-s} &= 344 \\
\text{t} &= 0.80 \times 10^0 & \text{no. steps} &= 307 & \text{no. f-s} &= 351 \\
\text{t} &= 0.90 \times 10^0 & \text{no. steps} &= 312 & \text{no. f-s} &= 357 \\
\text{t} &= 1.0 \times 10^1 & \text{no. steps} &= 317 & \text{no. f-s} &= 362 \\
\end{align*}
\]

Max. absolute error is 0.90E-08

Final statistics:

\[
\begin{align*}
\text{number of steps} &= 317 & \text{number of f evals.} &= 362 \\
\text{number of prec. setups} &= 33 \\
\text{number of prec. evals.} &= 6 & \text{number of prec. solves} &= 644 \\
\text{number of nonl. iters.} &= 359 & \text{number of lin. iters.} &= 322 \\
\text{average Krylov subspace dimension (NLI/NNI)} &= 0.8969 \\
\text{number of conv. failures... nonlinear} &= 0 & \text{linear} &= 0 \\
\text{number of error test failures} &= 5 \\
\text{main solver real/int workspace sizes} &= 349 92 \\
\text{linear solver real/int workspace sizes} &= 294 78 \\
\end{align*}
\]

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

The stiff example problem \texttt{cvDiurnal\_kry} described above, or rather its parallel version \texttt{cvDiurnal\_kry\_p}, has been modified and expanded to form a test problem for the parallel version of \texttt{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:

$$
\beta(y) = 0.75 + 0.25 \tanh(10y - 400).
$$

(11)

This function rises from about 0.5 to about 1.0 over a $y$ interval of about 0.2 (i.e. 1/100 of the total span in $y$). This vertical variation, together with the horizontal 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 $\partial c_i / \partial x_j$, namely:

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

(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 $\text{MXSUB} = \text{MYSUB} = 50$, so that the local (per-processor) problem size is 5000, while the processor array dimensions, $\text{NPEX}$ and $\text{NPEY}$, are varied. In one (typical) sequence of tests, we fix $\text{NPEY} = 8$ (for a vertical mesh size of $\text{MY} = 400$), and set $\text{NPEX} = 8$ ($\text{MX} = 400$), $\text{NPEX} = 16$ ($\text{MX} = 800$), and $\text{NPEX} = 32$ ($\text{MX} = 1600$). 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, $\text{maxl}$, 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, \texttt{nst} is the number of time steps, \texttt{nfe} is the number of $f$ evaluations, \texttt{nli} is the number of nonlinear (Newton) iterations, \texttt{nni} is the number of linear (Krylov) iterations, and \texttt{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 $\text{nli/nni}$, 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,
Table 1: Parallel cvode test results vs problem size and message-passing library

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

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

