Example Programs for CVODE v2.5.0

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

November 6, 2006

UCRL-SM-208110

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Contents

1 Introduction

This report is intended to serve as a companion document to the User Documentation of CVODE [1]. It provides details, with listings, on the example programs supplied with the CVODE distribution package.

The CVODE distribution contains examples of four types: serial C examples, parallel C examples, and serial and parallel FORTRAN examples. The following lists summarize all of these examples.

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

- `cvdenx` solves a chemical kinetics problem consisting of three rate equations.
  This program solves the problem with the BDF method and Newton iteration, with the CVDENSE linear solver and a user-supplied Jacobian routine. It also uses the rootfinding feature of CVODE.

- `cvdenx_uw` is the same as `cvdenx` but demonstrates the user-supplied error weight function feature of CVODE.

- `cvbanx` 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 CVBAND linear solver and a user-supplied Jacobian routine.

- `cvkryx` 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 CVSPGMR linear solver) 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.

- `cvkryx_bp` solves the same problem as `cvkryx`, 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.

- `cvkrydem_lin` solves the same problem as `cvkryx`, with the BDF method, but with three Krylov linear solvers: CVSPGMR, CVSPBCG, and CVSPTFQMR.

- `cvdirectdem` 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 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 approximation.

- `cvkrydem_pre` is a demonstration program for CVODE with the Krylov 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 and the CVSPGMR linear solver (scaled preconditioned GMRES).
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.

Supplied in the `sundials/examples/cvode/parallel` directory are the following three parallel examples (using the NVECTOR_PARALLEL module):

- `cvnonx_p` solves the semi-discrete form of an advection-diffusion equation in 1-D.
  This program solves the problem with the option for nonstiff systems, i.e. Adams method and functional iteration.
- `cvkryx_p` is the parallel implementation of `cvkryx`.
- `cvkryx_bbd_p` solves the same problem as `cvkryx_p`, with the BDF/GMRES method and a block-diagonal matrix with banded blocks as a preconditioner, generated by difference quotients, using the module CVBBDPRE.

With the FCVODE module, in the directories `sundials/examples/cvode/fcmix_serial` and `sundials/examples/cvode/fcmix_parallel`, are the following examples for the FORTRAN-C interface:

- `fcvdenx` is a serial chemical kinetics example (`BDF`/DENSE) with rootfinding.
- `fcvbanx` is a serial advection-diffusion example (`BDF`/BAND).
- `fcvkryx` is a serial kinetics-transport example (`BDF`/SPGMR).
- `fcvkryx_bp` is the `fcvkryx` example with FCVBP.
- `fcvnonx_p` is a parallel diagonal ODE example (`ADAMS`/`FUNCTIONAL`).
- `fcvkryx_p` is a parallel diagonal ODE example (`BDF`/SPGMR).
- `fcvkryx_bbd_p` is a parallel diagonal ODE example (`BDF`/SPGMR with FCVBBD).

In the following sections, we give detailed descriptions of some (but not all) of these examples. The Appendices contain complete listings of those examples described below. 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 `cvkryx`/`cvkryx_p` example.

In the descriptions below, we make frequent references to the CVODE User Document [1]. All citations to specific sections (e.g. §5.2) 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 §2). 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 variable `SUNDIALS_EXTENDED_PRECISION` to test if the solver libraries were built in extended precision and use the appropriate conversion specifiers in `printf` functions. Similarly, the FORTRAN examples in FCVODE are automatically pre-processed to generate source code that corresponds to the manner in which the CVODE libraries were built (see §4 in this document for more details).

2 Serial example problems

2.1 A dense example: cvdenx

As an initial illustration of the use of the CVODE package for the integration of IVP ODEs, we give a sample program called `cvdenx.c`. It uses the CVODE dense linear solver module CVDENSE 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, listed in Appendix A, 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 §5.2 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_dense.h` file provides the prototype for the `CVDense` function. 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. The `sundials_dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. We have explicitly included `sundials_dense.h`, but this is not necessary because it is included by `cvode_dense.h`.

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 `DenseMat`. It is defined using the DENSE accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macro `NV_Ith_S` is fully described in §7.1. The macro `DENSE_ELEM` is fully described in §5.6.4.

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 type `N_Vector`. 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 `CVodeMalloc` allocates 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 argument `CV_ SV` specifies a vector of absolute tolerances, and this is followed by the value of the relative tolerance `reltol` and the absolute tolerance vector `abstol`. See §5.5.1 for full details of this call.

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 §5.7.1 for a detailed description of this call.

The calls to `CVDense` (see §5.5.3) and `CVDenseSetJacFn` (see §5.5.5) specify the CVDENSE linear solver with an 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 §5.5.4 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`, and `CVodeFree` to free the CVODE memory block.

The function `PrintFinalStats` used here is actually suitable for general use in applications of CVODE to any problem with a dense Jacobian. It calls various `CVodeGet***` and `CVDenseGet***` functions to obtain the relevant counters, and then prints them. Specifically, 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 (`nfeD` = 0 here), the number of Jacobian evaluations (`njeD`), 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 §5.5.7.

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 §5.6.1 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 §5.7.2 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 `DenseMat`. Here the problem size is small, so we need not worry about the inefficiency of using `NV_Ith_S` and `DENSE_ELEM` to access `N_Vector` and `DenseMat` elements. Note that in this example, `Jac` only accesses the `y` and `J` arguments. See §5.6.4 for a detailed description of the dense `Jac` function.

The output generated by `cvdenx` 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.

cvdenx 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 =  7.157952e-01    9.183486e-06    2.841956e-01
At t = 4.0000e+02      y =  4.505420e-01    3.222963e-06    5.494548e-01
At t = 4.0000e+03      y =  1.831878e-01    8.941319e-07    8.168113e-01
At t = 4.0000e+04      y =  3.897868e-02    1.621567e-07    9.610212e-01
At t = 4.0000e+05      y =  4.940023e-03    1.985716e-08    9.950600e-01
At t = 4.0000e+06      y =  5.165107e-04    2.067097e-09    9.994835e-01
At t = 2.0807e+07      y =  1.000000e-04    4.000395e-10    9.999000e-01
    rootsfound[] =   1   0
At t = 4.0000e+07      y =  5.201457e-05    2.080690e-10    9.999480e-01
At t = 4.0000e+08      y =  5.207182e-06    2.082883e-11    9.999948e-01
At t = 4.0000e+09      y =  5.105811e-07    2.042325e-12    9.999995e-01
At t = 4.0000e+10      y =  4.511312e-08    1.804525e-13    1.000000e-00

Final Statistics:
nst = 515    nfe  = 755    nsetups = 110    nfeLS = 0      nje = 12
nni = 751    ncfn = 0      netf = 26     nge = 543

```

2.2 A banded example: cvbanx

The example program `cvbanx.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

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

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 $u_{(j-1)+(i-1)*\texttt{MY}} = v_{ij}$, 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 source is listed in Appendix B.

The cvbanx.c program includes files cvode_band.h and sundials_band.h in order to use the CVBAND linear solver. The cvode_band.h file contains the prototype for the CVBand routine. The sundials_band.h file contains the definition for band matrix type BandMat and the BAND_COL and BAND_COL_ELEM macros for accessing matrix elements (see §9.2). We have explicitly included sundials_band.h, but this is not necessary because it is included by cvode_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 both CVodeSetFData and CVBandSetJacFn, and as a result it will be passed back to the f and Jac functions each time they are called. (If appropriate, two different data structures could be defined and passed to f and Jac.) 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. In the CVodeMalloc call, the parameter SS indicates scalar relative and absolute tolerances, and pointers &reltol and &abstol to these values are passed. The call to CVBand (see §5.5.3) specifies the CVBAND linear solver, and specifies that both half-bandwidths of the Jacobian are equal to MY. The call to CVBandSetJacFn (see §5.5.5) specifies that a user-supplied Jacobian function Jac is to be used and that a pointer to data shold be passed to Jac every time it is called. 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 cvbanx.c file are definitions of five functions: f, Jac, SetIC, PrintFinalStats, and check_flag. The last three functions are called only from within the cvbanx.c file. The SetIC function sets the initial dependent variable vector; 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 (nfeB $= 0$ here), Jacobian evaluations (njeB), and nonlinear iterations (nni). These optional outputs are described in §5.5.7. Note that PrintFinalStats is suitable for general use in applications of CVODE to any problem with a banded Jacobian.

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 `f_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 `BAND_COL_ELEM` 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 `BAND_COL_ELEM`. The first argument passed to the `BAND_COL_ELEM` macro is a pointer to the diagonal element in the column to be accessed. This pointer is obtained via a call to the `BAND_COL` 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 `BAND_COL_ELEM` prevent illegal indexing. See §5.6.5 for a detailed description of the banded `Jac` function.

The output generated by `cvbanx` is shown below.

cvbanx 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: cvkryx

We give here an example that illustrates the use of CVODE with the Krylov method SPGMR, in the CVSPGMR module, as the linear system solver. The source file, `cvkryx.c`, is listed in Appendix C.

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

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

The spatial domain is $0 \leq x \leq 20$, $30 \leq y \leq 50$ (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:

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

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 `cvode_spgmr.h` and `sundials_math.h`. The first contains constants and function prototypes associated with the SPGMR method, including the values of the `pretype` argument to `CVSpgmr`. The inclusion of `sundials_math.h` is done to access the `SQR` 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 `CVodeMalloc` with scalar tolerances. It calls `CVSpgmr` (see §5.5.3) to specify the CVSPGMR linear solver with left preconditioning, and the default value (indicated by a zero argument) for `maxl`. The Gram-Schmidt orthogonalization is set to `MODIFIED_GS` through the function `CVSpilsSetGSType`. Next, user-supplied preconditioner setup and solve functions, `Precond` and `PSolve`, as well as the `data` pointer passed to `Precond` and `PSolve` whenever these are called, See §5.5.5 for details on the `CVSpilsSetPreconditioner` function.

Then 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 (`nfel`), 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 CVSPGMR, denoted `llrw` and `lliw`. All of these optional outputs are described in §5.5.7. The `PrintFinalStats` function is suitable for general use in applications of CVODE to any problem with the SPGMR 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 §5.6.8) and backsolved in the `PSolve` routine (see §5.6.7). 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 == FALSE`. On calls with `jok == TRUE`, 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 == TRUE` can only occur after a prior call with `jok == FALSE`.) The `Precond` routine must also set the value of `jcur`, i.e. `*jcurPtr`, to `TRUE` when J_{bd} is re-evaluated, and `FALSE` otherwise, to inform CVSPGMR of the status of Jacobian data.

We need to take a brief detour to explain one last important aspect of the `cvkryx.c` program. The generic DENSE solver contains two sets of functions: one for "large" matrices and one for "small" matrices. The large dense functions work with the type `DenseMat`, while the small dense functions work with `realtype **` as the underlying dense matrix types. The CVDENSE linear solver uses the type `DenseMat` for the $N \times N$ dense Jacobian and Newton matrices, and calls the large matrix functions. But to avoid the extra layer of function calls, `cvkryx.c` uses the small dense functions for all operations on the 2×2 preconditioner blocks. Thus it includes `sundials_smalldense.h`, and calls the small dense matrix functions `denalloc`, `dencopy`, `denscale`, `denaddI`, `denfree`, `denfreepiv`, `denGETRF`, and `denGETRS`. The macro `IJth` defined near the top of the file is used to access individual elements in each preconditioner block, numbered from 1. The small dense functions are available for CVODE user programs generally, and are documented in §9.1.

In addition to the functions called by CVODE, `cvkryx.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 `cvkryx.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.

cvkryx 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 = 301   order = 5   stepsize = 3.87e+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 = 343   order = 3   stepsize = 2.34e+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 = 421   order = 4   stepsize = 5.26e+02
c1 (bot.left/middle/top rt.) =   -4.385e-06    -1.528e-06    -4.905e-06
c2 (bot.left/middle/top rt.) =    3.382e+11     1.355e+11     3.804e+11

t = 5.04e+04   no. steps = 445   order = 3   stepsize = 1.98e+02
c1 (bot.left/middle/top rt.) =    4.461e-07     1.869e-07     4.842e-07
c2 (bot.left/middle/top rt.) =    3.358e+11     4.930e+11     3.864e+11

t = 5.76e+04   no. steps = 462   order = 5   stepsize = 2.35e+02
c1 (bot.left/middle/top rt.) =    3.204e-09     1.203e-09     3.555e-09
c2 (bot.left/middle/top rt.) =    3.320e+11     9.650e+11     3.909e+11

t = 6.48e+04   no. steps = 474   order = 5   stepsize = 6.02e+02
c1 (bot.left/middle/top rt.) =   -1.066e-09    -3.409e-10    -1.206e-09
c2 (bot.left/middle/top rt.) =    3.313e+11     8.922e+11     3.963e+11

t = 7.20e+04   no. steps = 486   order = 5   stepsize = 6.02e+02
c1 (bot.left/middle/top rt.) =    2.614e-09     9.722e-10     2.904e-09
c2 (bot.left/middle/top rt.) =    3.330e+11     6.186e+11     4.039e+11

t = 7.92e+04   no. steps = 498   order = 5   stepsize = 6.02e+02
c1 (bot.left/middle/top rt.) =    4.649e-11     1.729e-11     5.161e-11
c2 (bot.left/middle/top rt.) =    3.334e+11     6.669e+11     4.120e+11

t = 8.64e+04   no. steps = 510   order = 5   stepsize = 6.02e+02
c1 (bot.left/middle/top rt.) =   -8.856e-14    -3.348e-14    -9.785e-14
c2 (bot.left/middle/top rt.) =    3.352e+11     9.107e+11     4.163e+11


Final Statistics..
```

```
lenrw   =  2089     leniw   =    50
lenrwLS =  2046     leniwLS =    10
nst     =   510
nfe     =   675     nfeLS   =   641
nni     =   671     nli     =   641
nsetups =    94     netf    =    36
npe     =     9     nps     =  1243
ncfn    =     0     ncfl    =     0
```

3 Parallel example problems

3.1 A nonstiff example: cvnonx_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 $0 \leq t \leq 5, \quad 0 \leq x \leq 2$, 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.

The source file for this problem, `cvnonx_p.c`, is listed in Appendix D. It uses the Adams (non-stiff) integration formula and functional iteration. This problem is unrealistically simple, but serves to illustrate use of the parallel version of CVODE.

The `cvnonx_p.c` file begins with `#include` lines, 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` and `CVodeMalloc` specify a CVODE solution with the nonstiff method and scalar tolerances. The call to `CVodeSetFdata` 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 `cvnonx_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 `cvnonx_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.

cvnonx_p sample output

```

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

 Number of PEs =   4

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) =  7.109707e-02  nst = 339
At t = 2.50  max.norm(u) =  2.026223e-02  nst = 418
At t = 3.00  max.norm(u) =  5.777332e-03  nst = 486
At t = 3.50  max.norm(u) =  1.650483e-03  nst = 563
At t = 4.00  max.norm(u) =  4.754357e-04  nst = 646
At t = 4.50  max.norm(u) =  1.374222e-04  nst = 715
At t = 5.00  max.norm(u) =  3.937469e-05  nst = 795

Final Statistics:

nst = 795     nfe  = 1465    nni = 1461    ncfn = 146     netf = 5

```

3.2 A user preconditioner example: cvkryx_p

As an example of using CVODE with the Krylov linear solver CVSPGMR and the parallel MPI NVECTOR_PARALLEL module, we describe a test problem based on the system PDEs given above for the `cvkryx` 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.

The code listing for this example is given in Appendix E. The purpose of this code is to provide a more realistic example than that in `cvnonx_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 `cvkryx_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 `cvkryx_p.c` are the same as in `cvkryx.c`.

The following is a sample output from `cvkryx_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.

cvkryx_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 = 307   order = 4   stepsize = 1.98e+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 = 335   order = 5   stepsize = 1.17e+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 = 388   order = 4   stepsize = 4.48e+02
At bottom left:  c1, c2 =   -5.732e-07    3.382e+11
At top right:    c1, c2 =   -6.367e-07    3.804e+11

t = 5.04e+04   no. steps = 406   order = 5   stepsize = 3.97e+02
At bottom left:  c1, c2 =   -4.317e-09    3.358e+11
At top right:    c1, c2 =   -8.233e-09    3.864e+11

t = 5.76e+04   no. steps = 418   order = 5   stepsize = 4.74e+02
At bottom left:  c1, c2 =   -2.576e-09    3.320e+11
At top right:    c1, c2 =   -1.259e-09    3.909e+11

t = 6.48e+04   no. steps = 428   order = 5   stepsize = 7.70e+02
At bottom left:  c1, c2 =    3.451e-09    3.313e+11
At top right:    c1, c2 =    2.081e-09    3.963e+11

t = 7.20e+04   no. steps = 437   order = 5   stepsize = 7.70e+02
At bottom left:  c1, c2 =    1.630e-11    3.330e+11
At top right:    c1, c2 =    1.843e-11    4.039e+11

t = 7.92e+04   no. steps = 447   order = 5   stepsize = 7.70e+02
At bottom left:  c1, c2 =   -1.704e-11    3.334e+11
At top right:    c1, c2 =   -1.131e-11    4.120e+11

t = 8.64e+04   no. steps = 456   order = 5   stepsize = 7.70e+02
At bottom left:  c1, c2 =    1.496e-12    3.352e+11
At top right:    c1, c2 =    8.085e-13    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2046     leniwls =    80
nst     =   456
nfe     =   586     nfels   =   619
nni     =   582     nli     =   619
nsetups =    73     netf    =    25
npe     =     8     nps     =  1149
ncfn    =     0     ncfl    =     0
```

3.3 A CVBBDPRE preconditioner example: cvkryx_bbd_p

In this example, `cvkryx_bbd_p`, we solve the same problem in `cvkryx_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 `CVBBDPrecAlloc`. 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. The source is listed in Appendix F.

As in `cvkryx_p.c`, the `f` routine in `cvkryx_bbd_p.c` simply calls a

communication routine, `fucomm`, and then a strictly computational routine, `flocal`. However, the call to `CVBBDPrecAlloc` specifies the pair of routines to be called as `ucomm` and `flocal`, where `ucomm` is an *empty* routine. 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 `cvkryx_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 (`CVSpilsSetPrecType`).

Sample output from `cvkryx_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 `cvkryx_p` when preconditioning is on the left, but slightly poorer when it is on the right.

cvkryx_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 = 4.05e+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 = 309   order = 4   stepsize = 7.53e+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 = 377   order = 4   stepsize = 4.02e+02
At bottom left:  c1, c2 =    1.908e-07    3.382e+11
At top right:    c1, c2 =    2.345e-07    3.804e+11

t = 5.04e+04   no. steps = 392   order = 5   stepsize = 3.67e+02
At bottom left:  c1, c2 =   -6.408e-10    3.358e+11
At top right:    c1, c2 =   -6.654e-10    3.864e+11

t = 5.76e+04   no. steps = 403   order = 5   stepsize = 4.72e+02
At bottom left:  c1, c2 =    2.017e-08    3.320e+11
At top right:    c1, c2 =    3.353e-08    3.909e+11

t = 6.48e+04   no. steps = 415   order = 5   stepsize = 7.47e+02
At bottom left:  c1, c2 =   -2.502e-10    3.313e+11
At top right:    c1, c2 =    2.005e-10    3.963e+11

t = 7.20e+04   no. steps = 424   order = 5   stepsize = 7.47e+02
At bottom left:  c1, c2 =    4.217e-12    3.330e+11
At top right:    c1, c2 =   -2.693e-12    4.039e+11

t = 7.92e+04   no. steps = 434   order = 5   stepsize = 7.47e+02
At bottom left:  c1, c2 =    2.779e-12    3.334e+11
At top right:    c1, c2 =   -1.865e-12    4.120e+11

t = 8.64e+04   no. steps = 444   order = 5   stepsize = 7.47e+02
At bottom left:  c1, c2 =    2.331e-13    3.352e+11
At top right:    c1, c2 =   -1.599e-13    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2046     leniwls =    80
nst     =   444
nfe     =   581     nfels   =   526
nni     =   577     nli     =   526
nsetups =    75     netf    =    28
npe     =     8     nps     =  1057
ncfn    =     0     ncfl    =     0

In CVBBDPRE: real/integer local work space sizes = 600, 50
             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 = 314   order = 3   stepsize = 9.38e+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 = 350   order = 5   stepsize = 9.78e+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 = 403   order = 4   stepsize = 3.87e+02
At bottom left:  c1, c2 =    1.504e-09    3.382e+11
At top right:    c1, c2 =    1.683e-09    3.804e+11

t = 5.04e+04   no. steps = 416   order = 5   stepsize = 5.91e+02
At bottom left:  c1, c2 =   -1.137e-11    3.358e+11
At top right:    c1, c2 =   -1.439e-11    3.864e+11

t = 5.76e+04   no. steps = 432   order = 5   stepsize = 1.73e+02
At bottom left:  c1, c2 =    1.293e-09    3.320e+11
At top right:    c1, c2 =    2.448e-10    3.909e+11

t = 6.48e+04   no. steps = 447   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =    7.963e-13    3.313e+11
At top right:    c1, c2 =   -2.943e-13    3.963e+11

t = 7.20e+04   no. steps = 459   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =   -2.414e-12    3.330e+11
At top right:    c1, c2 =    2.797e-13    4.039e+11

t = 7.92e+04   no. steps = 470   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =   -1.059e-13    3.334e+11
At top right:    c1, c2 =    3.557e-14    4.120e+11

t = 8.64e+04   no. steps = 481   order = 5   stepsize = 6.38e+02
At bottom left:  c1, c2 =    6.045e-15    3.352e+11
At top right:    c1, c2 =   -2.016e-15    4.163e+11


Final Statistics:

lenrw   =  2089     leniw   =   120
lenrwls =  2046     leniwls =    80
nst     =   481
nfe     =   622     nfels   =   769
nni     =   618     nli     =   769
nsetups =   104     netf    =    33
npe     =     9     nps     =  1281
ncfn    =     0     ncfl    =     0

In CVBBDPRE: real/integer local work space sizes = 600, 50
             no. flocal evals. = 198
```

4 Fortran example problems

The FORTRAN example problem programs supplied with the CVODE package are all written in standard FORTRAN77 and use double-precision arithmetic. However, when the FORTRAN examples are built, the source code is automatically modified according to the configure options supplied by the user and the system type. Integer variables are declared as `INTEGER*`n, where n denotes the number of bytes in the corresponding C type (`long int` or `int`). Floating-point variable declarations remain unchanged if double-precision is used, but are changed to `REAL*`n, where n denotes the number of bytes in the SUNDIALS type `realtype`, if using single-precision. Also, if using single-precision, then declarations of floating-point constants are appropriately modified; e.g. `0.5D-4` is changed to `0.5E-4`.

4.1 A serial example: fcvkryx

The `fcvkryx` example is a FORTRAN equivalent of the `cvkryx` problem. (In fact, it was derived from an earlier FORTRAN example program for VODPK.) The source program `fcvkryx.f` is listed in Appendix G.

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` with its initial values. It calls `FNVINITS`, `FCVMALLOC`, `FCVSPGMR`, `FCVSPGMRSETPSET`, and `FCVSPGMRSETPSOL` to initialize the NVECTOR_SERIAL module, the main solver memory, and the CVSPGMR module, 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 `fcvkryx.f`, the `FCVFUN` routine is a straghtforward 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 $\texttt{JOK} = 0$, but is simply copied from `BD` to `P` if $\texttt{JOK} = 1$. 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. The remainder of `fcvkryx.f` consists of routines from LINPACK and the BLAS needed for matrix and vector operations.

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

fcvkryx 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 =   0.274587E+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 =   312  q =  4  h =   0.367517E+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 =   350  q =  4  h =   0.683836E+02
  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  0.376517E+12

 t =   0.432E+05   nst =   407  q =  4  h =   0.383863E+03
  c1 (bot.left/middle/top rt.) =   0.803367E-06  0.363858E-06  0.889797E-06
  c2 (bot.left/middle/top rt.) =   0.338233E+12  0.135487E+12  0.380352E+12

 t =   0.504E+05   nst =   436  q =  3  h =   0.215343E+03
  c1 (bot.left/middle/top rt.) =   0.375001E-05  0.665499E-06  0.454113E-05
  c2 (bot.left/middle/top rt.) =   0.335816E+12  0.493028E+12  0.386445E+12

 t =   0.576E+05   nst =   454  q =  5  h =   0.428080E+03
  c1 (bot.left/middle/top rt.) =   0.112301E-08  0.194567E-09  0.136087E-08
  c2 (bot.left/middle/top rt.) =   0.332031E+12  0.964985E+12  0.390900E+12

 t =   0.648E+05   nst =   466  q =  5  h =   0.690422E+03
  c1 (bot.left/middle/top rt.) =   0.353041E-08  0.590752E-09  0.428410E-08
  c2 (bot.left/middle/top rt.) =   0.331303E+12  0.892184E+12  0.396342E+12

 t =   0.720E+05   nst =   476  q =  5  h =   0.690422E+03
  c1 (bot.left/middle/top rt.) =  -0.121418E-09 -0.206756E-10 -0.147240E-09
  c2 (bot.left/middle/top rt.) =   0.332972E+12  0.618620E+12  0.403885E+12

 t =   0.792E+05   nst =   487  q =  5  h =   0.690422E+03
  c1 (bot.left/middle/top rt.) =  -0.341376E-11 -0.568210E-12 -0.414339E-11
  c2 (bot.left/middle/top rt.) =   0.333441E+12  0.666893E+12  0.412026E+12

 t =   0.864E+05   nst =   497  q =  5  h =   0.690422E+03
  c1 (bot.left/middle/top rt.) =   0.309841E-12  0.526192E-13  0.375773E-12
  c2 (bot.left/middle/top rt.) =   0.335178E+12  0.910652E+12  0.416251E+12


Final statistics:

 number of steps        =   497      number of f evals.     =  651
 number of prec. setups =    91
 number of prec. evals. =     9      number of prec. solves =  1233
 number of nonl. iters. =   647      number of lin. iters.  =   652
 average Krylov subspace dimension (NLI/NNI)  =   0.100773E+01
 number of conv. failures.. nonlinear =   0 linear =   0
 number of error test failures =  34
```

4.2 A parallel example: fcvkryx_bbd_p

This example, `fcvkryx_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, `fcvkryx_bbd_p.f`, is listed in Appendix H. It begins with MPI calls to initialize MPI and to get the number of processors and local processor index. The linear solver specification is done with calls to `FCVBBDINIT` and `FCVBBDSPGMR`. 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 `FCVSPGMRREINIT`, 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 `fcvkryx_bbd_p`, with `NPES` $= 4$. As expected, the performance is identical for left vs right preconditioning.

fcvkryx_bbd_p sample output

```
Diagonal test problem:

 NEQ =  40
 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 =   4


Preconditioning on left

 t =   0.10E+00     no. steps =   221   no. f-s =   262
 t =   0.20E+00     no. steps =   265   no. f-s =   308
 t =   0.30E+00     no. steps =   290   no. f-s =   334
 t =   0.40E+00     no. steps =   306   no. f-s =   351
 t =   0.50E+00     no. steps =   319   no. f-s =   365
 t =   0.60E+00     no. steps =   329   no. f-s =   375
 t =   0.70E+00     no. steps =   339   no. f-s =   386
 t =   0.80E+00     no. steps =   345   no. f-s =   392
 t =   0.90E+00     no. steps =   352   no. f-s =   399
 t =   0.10E+01     no. steps =   359   no. f-s =   406

Max. absolute error is  0.28E-08


Final statistics:

 number of steps        =   359     number of f evals.     =   406
 number of prec. setups =    38
 number of prec. evals. =     7     number of prec. solves =   728
 number of nonl. iters. =   402     number of lin. iters.  =   364
 average Krylov subspace dimension (NLI/NNI) =   0.9055
 number of conv. failures.. nonlinear =   0  linear =   0
 number of error test failures =   5
 main solver real/int workspace sizes   =   489  120
 linear solver real/int workspace sizes =   446   80
In CVBBDPRE:
 real/int local workspace =    20   10
 number of g evals. =    14


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



Preconditioning on right

 t =   0.10E+00     no. steps =   221   no. f-s =   262
 t =   0.20E+00     no. steps =   265   no. f-s =   308
 t =   0.30E+00     no. steps =   290   no. f-s =   334
 t =   0.40E+00     no. steps =   306   no. f-s =   351
 t =   0.50E+00     no. steps =   319   no. f-s =   365
 t =   0.60E+00     no. steps =   329   no. f-s =   375
 t =   0.70E+00     no. steps =   339   no. f-s =   386
 t =   0.80E+00     no. steps =   345   no. f-s =   392
 t =   0.90E+00     no. steps =   352   no. f-s =   399
 t =   0.10E+01     no. steps =   359   no. f-s =   406

Max. absolute error is  0.28E-08


Final statistics:

 number of steps        =   359     number of f evals.     =   406
 number of prec. setups =    38
 number of prec. evals. =     7     number of prec. solves =   728
 number of nonl. iters. =   402     number of lin. iters.  =   364
 average Krylov subspace dimension (NLI/NNI) =   0.9055
 number of conv. failures.. nonlinear =   0  linear =   0
 number of error test failures =   5
 main solver real/int workspace sizes   =   489  120
 linear solver real/int workspace sizes =   446   80
In CVBBDPRE:
 real/int local workspace =    20   10
 number of g evals. =    14
```

5 Parallel tests

The stiff example problem `cvkryx` described above, or rather its parallel version `cvkry_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 [2].

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] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.4.0. Technical Report UCRL-SM-208108, LLNL, 2005.

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

A Listing of cvdenx.c

```c
/*
 * -----------------------------------------------------------------
 * $Revision: 1.1 $
 * $Date: 2006/07/05 15:50:05 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem, with the coding
 * needed for its solution by CVODE. The problem is from
 * chemical kinetics, and consists of the following three rate
 * equations:
 *    dy1/dt = -.04*y1 + 1.e4*y2*y3
 *    dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*(y2)^2
 *    dy3/dt = 3.e7*(y2)^2
 * on the interval from t = 0.0 to t = 4.e10, with initial
 * conditions: y1 = 1.0, y2 = y3 = 0. The problem is stiff.
 * While integrating the system, we also use the rootfinding
 * feature to find the points at which y1 = 1e-4 or at which
 * y3 = 0.01. This program solves the problem with the BDF method,
 * Newton iteration with the CVDENSE dense linear solver, and a
 * user-supplied Jacobian routine.
 * It uses a scalar relative tolerance and a vector absolute
 * tolerance. Output is printed in decades from t = .4 to t = 4.e10.
 * Run statistics (optional outputs) are printed at the end.
 * -----------------------------------------------------------------
 */

#include <stdio.h>

/* Header files with a description of contents used in cvdenx.c */

#include <cvode/cvode.h>             /* prototypes for CVODE fcts. and consts. */
#include <nvector/nvector_serial.h>  /* serial N_Vector types, fcts., and macros */
#include <cvode/cvode_dense.h>       /* prototype for CVDense */
#include <sundials/sundials_dense.h> /* definitions DenseMat DENSE_ELEM */
#include <sundials/sundials_types.h> /* definition of type realtype */

/* User-defined vector and matrix accessor macros: Ith, IJth */

/* These macros are defined in order to write code which exactly matches
   the mathematical problem description given above.

   Ith(v,i) references the ith component of the vector v, where i is in
   the range [1..NEQ] and NEQ is defined below. The Ith macro is defined
   using the N_VIth macro in nvector.h. N_VIth numbers the components of
   a vector starting from 0.

   IJth(A,i,j) references the (i,j)th element of the dense matrix A, where
   i and j are in the range [1..NEQ]. The IJth macro is defined using the
   DENSE_ELEM macro in dense.h. DENSE_ELEM numbers rows and columns of a
   dense matrix starting from 0. */

#define Ith(v,i)    NV_Ith_S(v,i-1)       /* Ith numbers components 1..NEQ */
#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */
```

```


/* Problem Constants */

#define NEQ   3                /* number of equations  */
#define Y1    RCONST(1.0)      /* initial y components */
#define Y2    RCONST(0.0)
#define Y3    RCONST(0.0)
#define RTOL  RCONST(1.0e-4)   /* scalar relative tolerance            */
#define ATOL1 RCONST(1.0e-8)   /* vector absolute tolerance components */
#define ATOL2 RCONST(1.0e-14)
#define ATOL3 RCONST(1.0e-6)
#define T0    RCONST(0.0)      /* initial time           */
#define T1    RCONST(0.4)      /* first output time      */
#define TMULT RCONST(10.0)     /* output time factor     */
#define NOUT  12               /* number of output times */


/* Functions Called by the Solver */

static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data);

static int g(realtype t, N_Vector y, realtype *gout, void *g_data);

static int Jac(long int N, DenseMat J, realtype t,
               N_Vector y, N_Vector fy, void *jac_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

/* Private functions to output results */

static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3);
static void PrintRootInfo(int root_f1, int root_f2);

/* Private function to print final statistics */

static void PrintFinalStats(void *cvode_mem);

/* Private function to check function return values */

static int check_flag(void *flagvalue, char *funcname, int opt);


/*
 *-------------------------------
 * Main Program
 *-------------------------------
 */

int main()
{
  realtype reltol, t, tout;
  N_Vector y, abstol;
  void *cvode_mem;
  int flag, flagr, iout;
  int rootsfound[2];

  y = abstol = NULL;
  cvode_mem = NULL;

  /* Create serial vector of length NEQ for I.C. and abstol */
  y = N_VNew_Serial(NEQ);
  if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
  abstol = N_VNew_Serial(NEQ);
  if (check_flag((void *)abstol, "N_VNew_Serial", 0)) return(1);

  /* Initialize y */
  Ith(y,1) = Y1;
  Ith(y,2) = Y2;
  Ith(y,3) = Y3;

  /* Set the scalar relative tolerance */
  reltol = RTOL;
  /* Set the vector absolute tolerance */
  Ith(abstol,1) = ATOL1;
  Ith(abstol,2) = ATOL2;
  Ith(abstol,3) = ATOL3;

  /*
     Call CVodeCreate to create the solver memory:

     CV_BDF     specifies the Backward Differentiation Formula
     CV_NEWTON  specifies a Newton iteration

     A pointer to the integrator problem memory is returned and stored in cvode_mem.
  */

  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if (check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

  /*
     Call CVodeMalloc to initialize the integrator memory:

     cvode_mem is the pointer to the integrator memory returned by CVodeCreate
     f         is the user's right hand side function in y'=f(t,y)
     T0        is the initial time
     y         is the initial dependent variable vector
     CV_SV     specifies scalar relative and vector absolute tolerances
     &reltol   is a pointer to the scalar relative tolerance
     abstol    is the absolute tolerance vector
  */

  flag = CVodeMalloc(cvode_mem, f, T0, y, CV_SV, reltol, abstol);
  if (check_flag(&flag, "CVodeMalloc", 1)) return(1);

  /* Call CVodeRootInit to specify the root function g with 2 components */
  flag = CVodeRootInit(cvode_mem, 2, g, NULL);
  if (check_flag(&flag, "CVodeRootInit", 1)) return(1);

  /* Call CVDense to specify the CVDENSE dense linear solver */
  flag = CVDense(cvode_mem, NEQ);
  if (check_flag(&flag, "CVDense", 1)) return(1);

  /* Set the Jacobian routine to Jac (user-supplied) */
  flag = CVDenseSetJacFn(cvode_mem, Jac, NULL);
  if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);

  /* In loop, call CVode, print results, and test for error.
     Break out of loop when NOUT preset output times have been reached.  */
  printf("␣\n3-species␣kinetics␣problem\n\n");

  iout = 0;  tout = T1;
  while(1) {
    flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
    PrintOutput(t, Ith(y,1), Ith(y,2), Ith(y,3));

    if (flag == CV_ROOT_RETURN) {
      flagr = CVodeGetRootInfo(cvode_mem, rootsfound);
      if (check_flag(&flagr, "CVodeGetRootInfo", 1)) return(1);
      PrintRootInfo(rootsfound[0],rootsfound[1]);
    }

    if (check_flag(&flag, "CVode", 1)) break;
    if (flag == CV_SUCCESS) {
      iout++;
      tout *= TMULT;
    }

    if (iout == NOUT) break;
  }

  /* Print some final statistics */
  PrintFinalStats(cvode_mem);

  /* Free y vector */
  N_VDestroy_Serial(y);

  /* Free integrator memory */
  CVodeFree(&cvode_mem);

  return(0);
}


/*
 *-------------------------------
 * Functions called by the solver
 *-------------------------------
 */

/*
 * f routine. Compute function f(t,y).
 */

static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
{
  realtype y1, y2, y3, yd1, yd3;

  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

  yd1 = Ith(ydot,1) = RCONST(-0.04)*y1 + RCONST(1.0e4)*y2*y3;
  yd3 = Ith(ydot,3) = RCONST(3.0e7)*y2*y2;
        Ith(ydot,2) = -yd1 - yd3;

  return(0);
}

/*
```

```
 * g routine. Compute functions g_i(t,y) for i = 0,1.
 */

static int g(realtype t, N_Vector y, realtype *gout, void *g_data)
{
  realtype y1, y3;

  y1 = Ith(y,1); y3 = Ith(y,3);
  gout[0] = y1 - RCONST(0.0001);
  gout[1] = y3 - RCONST(0.01);

  return(0);
}

/*
 * Jacobian routine. Compute J(t,y) = df/dy. *
 */

static int Jac(long int N, DenseMat J, realtype t,
               N_Vector y, N_Vector fy, void *jac_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  realtype y1, y2, y3;

  y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);

  IJth(J,1,1) = RCONST(-0.04);
  IJth(J,1,2) = RCONST(1.0e4)*y3;
  IJth(J,1,3) = RCONST(1.0e4)*y2;
  IJth(J,2,1) = RCONST(0.04);
  IJth(J,2,2) = RCONST(-1.0e4)*y3-RCONST(6.0e7)*y2;
  IJth(J,2,3) = RCONST(-1.0e4)*y2;
  IJth(J,3,2) = RCONST(6.0e7)*y2;

  return(0);
}

/*
 *-------------------------------
 * Private helper functions
 *-------------------------------
 */

static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3)
{
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("At t = %0.4Le      y =%14.6Le  %14.6Le  %14.6Le\n", t, y1, y2, y3);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("At t = %0.4le      y =%14.6le  %14.6le  %14.6le\n", t, y1, y2, y3);
#else
  printf("At t = %0.4e      y =%14.6e  %14.6e  %14.6e\n", t, y1, y2, y3);
#endif

  return;
}

static void PrintRootInfo(int root_f1, int root_f2)
{
  printf("    rootsfound[] = %3d %3d\n", root_f1, root_f2);

  return;
}

/*
 * Get and print some final statistics
 */

static void PrintFinalStats(void *cvode_mem)
{
  long int nst, nfe, nsetups, nje, nfeLS, nni, ncfn, netf, nge;
  int flag;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);

  flag = CVDenseGetNumJacEvals(cvode_mem, &nje);
  check_flag(&flag, "CVDenseGetNumJacEvals", 1);
  flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeLS);
  check_flag(&flag, "CVDenseGetNumRhsEvals", 1);

  flag = CVodeGetNumGEvals(cvode_mem, &nge);
  check_flag(&flag, "CVodeGetNumGEvals", 1);

  printf("\nFinal␣Statistics:\n");
  printf("nst␣=␣%-6ld␣nfe␣␣=␣%-6ld␣nsetups␣=␣%-6ld␣nfeLS␣=␣%-6ld␣nje␣=␣%ld\n",
         nst, nfe, nsetups, nfeLS, nje);
  printf("nni␣=␣%-6ld␣ncfn␣=␣%-6ld␣netf␣=␣%-6ld␣nge␣=␣%ld\n␣\n",
         nni, ncfn, netf, nge);
}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR:␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}
```

B Listing of cvbanx.c

```c
/*
 * -----------------------------------------------------------------
 * $Revision: 1.1 $
 * $Date: 2006/07/05 15:50:05 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem with a banded Jacobian,
 * with the program for its solution by CVODE.
 * The problem is the semi-discrete form of the advection-diffusion
 * equation in 2-D:
 *   du/dt = d^2 u / dx^2 + .5 du/dx + d^2 u / dy^2
 * on the rectangle 0 <= x <= 2, 0 <= y <= 1, and the time
 * interval 0 <= t <= 1. Homogeneous Dirichlet boundary conditions
 * are posed, and the initial condition is
 *   u(x,y,t=0) = x(2-x)y(1-y)exp(5xy).
 * The PDE is discretized on a uniform MX+2 by MY+2 grid with
 * central differencing, and with boundary values eliminated,
 * leaving an ODE system of size NEQ = MX*MY.
 * This program solves the problem with the BDF method, Newton
 * iteration with the CVBAND band linear solver, and a user-supplied
 * Jacobian routine.
 * It uses scalar relative and absolute tolerances.
 * Output is printed at t = .1, .2, ..., 1.
 * Run statistics (optional outputs) are printed at the end.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Header files with a description of contents used in cvbanx.c */

#include <cvode/cvode.h>             /* prototypes for CVODE fcts. and consts. */
#include <cvode/cvode_band.h>        /* prototype for CVBand */
#include <nvector/nvector_serial.h>  /* serial N_Vector types, fcts., and macros */
#include <sundials/sundials_band.h>  /* definitions of type BandMat and macros */
#include <sundials/sundials_types.h> /* definition of type realtype */
#include <sundials/sundials_math.h>  /* definition of ABS and EXP */

/* Problem Constants */

#define XMAX  RCONST(2.0)    /* domain boundaries         */
#define YMAX  RCONST(1.0)
#define MX    10             /* mesh dimensions           */
#define MY    5
#define NEQ   MX*MY          /* number of equations       */
#define ATOL  RCONST(1.0e-5) /* scalar absolute tolerance */
#define T0    RCONST(0.0)    /* initial time              */
#define T1    RCONST(0.1)    /* first output time         */
#define DTOUT RCONST(0.1)    /* output time increment     */
#define NOUT  10             /* number of output times    */

```

```c
#define ZERO RCONST(0.0)
#define HALF RCONST(0.5)
#define ONE  RCONST(1.0)
#define TWO  RCONST(2.0)
#define FIVE RCONST(5.0)

/* User-defined vector access macro IJth */

/* IJth is defined in order to isolate the translation from the
   mathematical 2-dimensional structure of the dependent variable vector
   to the underlying 1-dimensional storage.
   IJth(vdata,i,j) references the element in the vdata array for
   u at mesh point (i,j), where 1 <= i <= MX, 1 <= j <= MY.
   The vdata array is obtained via the macro call vdata = NV_DATA_S(v),
   where v is an N_Vector.
   The variables are ordered by the y index j, then by the x index i. */

#define IJth(vdata,i,j) (vdata[(j-1) + (i-1)*MY])

/* Type : UserData (contains grid constants) */

typedef struct {
  realtype dx, dy, hdcoef, hacoef, vdcoef;
} *UserData;

/* Private Helper Functions */

static void SetIC(N_Vector u, UserData data);
static void PrintHeader(realtype reltol, realtype abstol, realtype umax);
static void PrintOutput(realtype t, realtype umax, long int nst);
static void PrintFinalStats(void *cvode_mem);

/* Private function to check function return values */

static int check_flag(void *flagvalue, char *funcname, int opt);

/* Functions Called by the Solver */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
static int Jac(long int N, long int mu, long int ml, BandMat J,
               realtype t, N_Vector u, N_Vector fu, void *jac_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

/*
 *-------------------------------
 * Main Program
 *-------------------------------
 */

int main(void)
{
  realtype dx, dy, reltol, abstol, t, tout, umax;
  N_Vector u;
  UserData data;
  void *cvode_mem;
  int iout, flag;
  long int nst;

  u = NULL;
  data = NULL;
  cvode_mem = NULL;

  /* Create a serial vector */

  u = N_VNew_Serial(NEQ);  /* Allocate u vector */
  if(check_flag((void*)u, "N_VNew_Serial", 0)) return(1);

  reltol = ZERO;  /* Set the tolerances */
  abstol = ATOL;

  data = (UserData) malloc(sizeof *data);  /* Allocate data memory */
  if(check_flag((void *)data, "malloc", 2)) return(1);
  dx = data->dx = XMAX/(MX+1);  /* Set grid coefficients in data */
  dy = data->dy = YMAX/(MY+1);
  data->hdcoef = ONE/(dx*dx);
  data->hacoef = HALF/(TWO*dx);
  data->vdcoef = ONE/(dy*dy);

  SetIC(u, data);  /* Initialize u vector */

  /*
     Call CvodeCreate to create integrator memory

     CV_BDF     specifies the Backward Differentiation Formula
     CV_NEWTON  specifies a Newton iteration

     A pointer to the integrator problem memory is returned and
     stored in cvode_mem.
  */

  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

  /*
     Call CVodeMalloc to initialize the integrator memory:

     cvode_mem is the pointer to the integrator memory returned by CVodeCreate
     f       is the user's right hand side function in y'=f(t,y)
     T0      is the initial time
     u       is the initial dependent variable vector
     CV_SS   specifies scalar relative and absolute tolerances
     reltol  is the scalar relative tolerance
     &abstol is a pointer to the scalar absolute tolerance
  */

  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if(check_flag(&flag, "CVodeMalloc", 1)) return(1);

  /* Set the pointer to user-defined data */

  flag = CVodeSetFdata(cvode_mem, data);
  if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);

  /* Call CVBand to specify the CVBAND band linear solver */

  flag = CVBand(cvode_mem, NEQ, MY, MY);
  if(check_flag(&flag, "CVBand", 1)) return(1);

  /* Set the user-supplied Jacobian routine Jac and
     the pointer to the user-defined block data. */

  flag = CVBandSetJacFn(cvode_mem, Jac, data);
  if(check_flag(&flag, "CVBandSetJacFn", 1)) return(1);

  /* In loop over output points: call CVode, print results, test for errors */

  umax = N_VMaxNorm(u);
  PrintHeader(reltol, abstol, umax);
  for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    if(check_flag(&flag, "CVode", 1)) break;
    umax = N_VMaxNorm(u);
    flag = CVodeGetNumSteps(cvode_mem, &nst);
    check_flag(&flag, "CVodeGetNumSteps", 1);
    PrintOutput(t, umax, nst);
  }

  PrintFinalStats(cvode_mem);  /* Print some final statistics   */

  N_VDestroy_Serial(u);   /* Free the u vector */
  CVodeFree(&cvode_mem);  /* Free the integrator memory */
  free(data);             /* Free the user data */

  return(0);
}

/*
 *-------------------------------
 * Functions called by the solver
 *-------------------------------
 */

/* f routine. Compute f(t,u). */

static int f(realtype t, N_Vector u,N_Vector udot, void *f_data)
{
  realtype uij, udn, uup, ult, urt, hordc, horac, verdc, hdiff, hadv, vdiff;
  realtype *udata, *dudata;
  int i, j;
  UserData data;

  udata = NV_DATA_S(u);
  dudata = NV_DATA_S(udot);

  /* Extract needed constants from data */

  data = (UserData) f_data;
  hordc = data->hdcoef;
  horac = data->hacoef;
  verdc = data->vdcoef;

  /* Loop over all grid points. */

  for (j=1; j <= MY; j++) {

    for (i=1; i <= MX; i++) {

      /* Extract u at x_i, y_j and four neighboring points */

      uij = IJth(udata, i, j);
      udn = (j == 1)  ? ZERO : IJth(udata, i, j-1);
      uup = (j == MY) ? ZERO : IJth(udata, i, j+1);
      ult = (i == 1)  ? ZERO : IJth(udata, i-1, j);
      urt = (i == MX) ? ZERO : IJth(udata, i+1, j);

      /* Set diffusion and advection terms and load into udot */

      hdiff = hordc*(ult - TWO*uij + urt);
      hadv = horac*(urt - ult);
      vdiff = verdc*(uup - TWO*uij + udn);
      IJth(dudata, i, j) = hdiff + hadv + vdiff;
    }
  }

  return(0);
}

/* Jacobian routine. Compute J(t,u). */

static int Jac(long int N, long int mu, long int ml, BandMat J,
               realtype t, N_Vector u, N_Vector fu, void *jac_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  long int i, j, k;
  realtype *kthCol, hordc, horac, verdc;
  UserData data;

  /*
    The components of f = udot that depend on u(i,j) are
    f(i,j), f(i-1,j), f(i+1,j), f(i,j-1), f(i,j+1), with
      df(i,j)/du(i,j) = -2 (1/dx^2 + 1/dy^2)
      df(i-1,j)/du(i,j) = 1/dx^2 + .25/dx  (if i > 1)
      df(i+1,j)/du(i,j) = 1/dx^2 - .25/dx  (if i < MX)
      df(i,j-1)/du(i,j) = 1/dy^2           (if j > 1)
      df(i,j+1)/du(i,j) = 1/dy^2           (if j < MY)
  */

  data = (UserData) jac_data;
  hordc = data->hdcoef;
  horac = data->hacoef;
  verdc = data->vdcoef;

  for (j=1; j <= MY; j++) {
    for (i=1; i <= MX; i++) {
      k = j-1 + (i-1)*MY;
      kthCol = BAND_COL(J,k);

      /* set the kth column of J */

      BAND_COL_ELEM(kthCol,k,k) = -TWO*(verdc+hordc);
      if (i != 1)  BAND_COL_ELEM(kthCol,k-MY,k) = hordc + horac;
      if (i != MX) BAND_COL_ELEM(kthCol,k+MY,k) = hordc - horac;
      if (j != 1)  BAND_COL_ELEM(kthCol,k-1,k)  = verdc;
      if (j != MY) BAND_COL_ELEM(kthCol,k+1,k)  = verdc;
    }
  }
```

```c

  return(0);
}

/*
 *-------------------------------
 * Private helper functions
 *-------------------------------
 */

/* Set initial conditions in u vector */

static void SetIC(N_Vector u, UserData data)
{
  int i, j;
  realtype x, y, dx, dy;
  realtype *udata;

  /* Extract needed constants from data */

  dx = data->dx;
  dy = data->dy;

  /* Set pointer to data array in vector u. */

  udata = NV_DATA_S(u);

  /* Load initial profile into u vector */

  for (j=1; j <= MY; j++) {
    y = j*dy;
    for (i=1; i <= MX; i++) {
      x = i*dx;
      IJth(udata,i,j) = x*(XMAX - x)*y*(YMAX - y)*EXP(FIVE*x*y);
    }
  }
}

/* Print first lines of output (problem description) */

static void PrintHeader(realtype reltol, realtype abstol, realtype umax)
{
  printf("\n2-D Advection-Diffusion Equation\n");
  printf("Mesh dimensions = %d X %d\n", MX, MY);
  printf("Total system size = %d\n", NEQ);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters: reltol = %Lg   abstol = %Lg\n\n", reltol, abstol);
  printf("At t = %Lg       max.norm(u) =%14.6Le \n", T0, umax);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters: reltol = %lg   abstol = %lg\n\n", reltol, abstol);
  printf("At t = %lg       max.norm(u) =%14.6le \n", T0, umax);
#else
  printf("Tolerance parameters: reltol = %g   abstol = %g\n\n", reltol, abstol);
  printf("At t = %g       max.norm(u) =%14.6e \n", T0, umax);
#endif

  return;
}

```

```
/* Print current value */

static void PrintOutput(realtype t, realtype umax, long int nst)
{
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("At␣t␣=␣%4.2Lf␣␣␣max.norm(u)␣=%14.6Le␣␣␣nst␣=␣%4ld\n", t, umax, nst);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("At␣t␣=␣%4.2f␣␣␣max.norm(u)␣=%14.6le␣␣␣nst␣=␣%4ld\n", t, umax, nst);
#else
  printf("At␣t␣=␣%4.2f␣␣␣max.norm(u)␣=%14.6e␣␣␣nst␣=␣%4ld\n", t, umax, nst);
#endif

  return;
}

/* Get and print some final statistics */

static void PrintFinalStats(void *cvode_mem)
{
  int flag;
  long int nst, nfe, nsetups, netf, nni, ncfn, nje, nfeLS;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);

  flag = CVBandGetNumJacEvals(cvode_mem, &nje);
  check_flag(&flag, "CVBandGetNumJacEvals", 1);
  flag = CVBandGetNumRhsEvals(cvode_mem, &nfeLS);
  check_flag(&flag, "CVBandGetNumRhsEvals", 1);

  printf("\nFinal␣Statistics:\n");
  printf("nst␣=␣%-6ld␣nfe␣␣=␣%-6ld␣nsetups␣=␣%-6ld␣nfeLS␣=␣%-6ld␣nje␣=␣%ld\n",
         nst, nfe, nsetups, nfeLS, nje);
  printf("nni␣=␣%-6ld␣ncfn␣=␣%-6ld␣netf␣=␣%ld\n␣\n",
         nni, ncfn, netf);

  return;
}

/* Check function return value...
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns a flag so check if
              flag >= 0
     opt == 2 means function allocates memory so check if returned
              NULL pointer */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */

  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */

  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */

  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}
```

C Listing of cvkryx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.2 $
 * $Date: 2006/10/11 16:33:56 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * An ODE system is generated from the following 2-species diurnal
 * kinetics advection-diffusion PDE system in 2 space dimensions:
 *
 * dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)
 *                 + Ri(c1,c2,t)      for i = 1,2,   where
 *   R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,
 *   R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,
 *   Kv(y) = Kv0*exp(y/5) ,
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)
 * vary diurnally. The problem is posed on the square
 *   0 <= x <= 20,    30 <= y <= 50   (all in km),
 * with homogeneous Neumann boundary conditions, and for time t in
 *   0 <= t <= 86400 sec (1 day).
 * The PDE system is treated by central differences on a uniform
 * 10 x 10 mesh, with simple polynomial initial profiles.
 * The problem is solved with CVODE, with the BDF/GMRES
 * method (i.e. using the CVSPGMR linear solver) 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 Precond
 * routine.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <cvode/cvode.h>                 /* main integrator header file */
#include <cvode/cvode_spgmr.h>           /* prototypes & constants for CVSPGMR solver */
#include <nvector/nvector_serial.h>      /* serial N_Vector types, fct. and macros */
#include <sundials/sundials_smalldense.h> /* use generic DENSE solver in preconditioning */
#include <sundials/sundials_types.h>     /* definition of realtype */
#include <sundials/sundials_math.h>      /* contains the macros ABS, SQR, and EXP */

/* Problem Constants */

#define ZERO RCONST(0.0)
#define ONE  RCONST(1.0)
#define TWO  RCONST(2.0)

#define NUM_SPECIES  2                 /* number of species         */
#define KH           RCONST(4.0e-6)    /* horizontal diffusivity Kh */
#define VEL          RCONST(0.001)     /* advection velocity V      */
#define KV0          RCONST(1.0e-8)    /* coefficient in Kv(y)      */
#define Q1           RCONST(1.63e-16)  /* coefficients q1, q2, c3   */
#define Q2           RCONST(4.66e-16)
```

```
#define C3           RCONST(3.7e16)
#define A3           RCONST(22.62)     /* coefficient in expression for q3(t) */
#define A4           RCONST(7.601)     /* coefficient in expression for q4(t) */
#define C1_SCALE     RCONST(1.0e6)     /* coefficients in initial profiles
*/
#define C2_SCALE     RCONST(1.0e12)

#define T0           ZERO                 /* initial time */
#define NOUT         12                   /* number of output times */
#define TWOHR        RCONST(7200.0)       /* number of seconds in two hours  */
#define HALFDAY      RCONST(4.32e4)       /* number of seconds in a half day */
#define PI       RCONST(3.1415926535898)  /* pi */

#define XMIN         ZERO                 /* grid boundaries in x  */
#define XMAX         RCONST(20.0)
#define YMIN         RCONST(30.0)         /* grid boundaries in y  */
#define YMAX         RCONST(50.0)
#define XMID         RCONST(10.0)         /* grid midpoints in x,y */
#define YMID         RCONST(40.0)

#define MX           10             /* MX = number of x mesh points */
#define MY           10             /* MY = number of y mesh points */
#define NSMX         20             /* NSMX = NUM_SPECIES*MX */
#define MM           (MX*MY)        /* MM = MX*MY */

/* CVodeMalloc Constants */

#define RTOL    RCONST(1.0e-5)    /* scalar relative tolerance */
#define FLOOR   RCONST(100.0)     /* value of C1 or C2 at which tolerances */
                                  /* change from relative to absolute      */
#define ATOL    (RTOL*FLOOR)      /* scalar absolute tolerance */
#define NEQ     (NUM_SPECIES*MM)  /* NEQ = number of equations */

/* User-defined vector and matrix accessor macros: IJKth, IJth */

/* IJKth is defined in order to isolate the translation from the
   mathematical 3-dimensional structure of the dependent variable vector
   to the underlying 1-dimensional storage. IJth is defined in order to
   write code which indexes into small dense matrices with a (row,column)
   pair, where 1 <= row, column <= NUM_SPECIES.

   IJKth(vdata,i,j,k) references the element in the vdata array for
   species i at mesh point (j,k), where 1 <= i <= NUM_SPECIES,
   0 <= j <= MX-1, 0 <= k <= MY-1. The vdata array is obtained via
   the macro call vdata = NV_DATA_S(v), where v is an N_Vector.
   For each mesh point (j,k), the elements for species i and i+1 are
   contiguous within vdata.

   IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
   where 1 <= i,j <= NUM_SPECIES. The small matrix routines in dense.h
   work with matrices stored by column in a 2-dimensional array. In C,
   arrays are indexed starting at 0, not 1. */

#define IJKth(vdata,i,j,k) (vdata[i-1 + (j)*NUM_SPECIES + (k)*NSMX])
#define IJth(a,i,j)        (a[j-1][i-1])

/* Type : UserData
   contains preconditioner blocks, pivot arrays, and problem constants */

```

```c
typedef struct {
  realtype **P[MX][MY], **Jbd[MX][MY];
  long int *pivot[MX][MY];
  realtype q4, om, dx, dy, hdco, haco, vdco;
} *UserData;

/* Private Helper Functions */

static UserData AllocUserData(void);
static void InitUserData(UserData data);
static void FreeUserData(UserData data);
static void SetInitialProfiles(N_Vector u, realtype dx, realtype dy);
static void PrintOutput(void *cvode_mem, N_Vector u, realtype t);
static void PrintFinalStats(void *cvode_mem);
static int check_flag(void *flagvalue, char *funcname, int opt);

/* Functions Called by the Solver */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);

static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr, realtype gamma,
                   void *P_data, N_Vector vtemp1, N_Vector vtemp2,
                   N_Vector vtemp3);

static int PSolve(realtype tn, N_Vector u, N_Vector fu,
                  N_Vector r, N_Vector z,
                  realtype gamma, realtype delta,
                  int lr, void *P_data, N_Vector vtemp);


/*
 *-------------------------------
 * Main Program
 *-------------------------------
 */

int main()
{
  realtype abstol, reltol, t, tout;
  N_Vector u;
  UserData data;
  void *cvode_mem;
  int iout, flag;

  u = NULL;
  data = NULL;
  cvode_mem = NULL;

  /* Allocate memory, and set problem data, initial values, tolerances */
  u = N_VNew_Serial(NEQ);
  if(check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
  data = AllocUserData();
  if(check_flag((void *)data, "AllocUserData", 2)) return(1);
  InitUserData(data);
  SetInitialProfiles(u, data->dx, data->dy);
  abstol=ATOL;
  reltol=RTOL;

  /* Call CvodeCreate to create the solver memory

     CV_BDF     specifies the Backward Differentiation Formula
     CV_NEWTON  specifies a Newton iteration

     A pointer to the integrator memory is returned and stored in cvode_mem. */
  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

  /* Set the pointer to user-defined data */
  flag = CVodeSetFdata(cvode_mem, data);
  if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);

  /* Call CVodeMalloc to initialize the integrator memory:

     f       is the user's right hand side function in u'=f(t,u)
     T0      is the initial time
     u       is the initial dependent variable vector
     CV_SS   specifies scalar relative and absolute tolerances
     reltol  is the relative tolerance
     &abstol is a pointer to the scalar absolute tolerance      */
  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if(check_flag(&flag, "CVodeMalloc", 1)) return(1);

  /* Call CVSpgmr to specify the linear solver CVSPGMR
     with left preconditioning and the maximum Krylov dimension maxl */
  flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
  if(check_flag(&flag, "CVSpgmr", 1)) return(1);

  /* Set modified Gram-Schmidt orthogonalization, preconditioner
     setup and solve routines Precond and PSolve, and the pointer
     to the user-defined block data */
  flag = CVSpilsSetGSType(cvode_mem, MODIFIED_GS);
  if(check_flag(&flag, "CVSpilsSetGSType", 1)) return(1);

  flag = CVSpilsSetPreconditioner(cvode_mem, Precond, PSolve, data);
  if(check_flag(&flag, "CVSpilsSetPreconditioner", 1)) return(1);

  /* In loop over output points, call CVode, print results, test for error */
  printf("␣\n2-species␣diurnal␣advection-diffusion␣problem\n\n");
  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    PrintOutput(cvode_mem, u, t);
    if(check_flag(&flag, "CVode", 1)) break;
  }

  PrintFinalStats(cvode_mem);

  /* Free memory */
  N_VDestroy_Serial(u);
  FreeUserData(data);
  CVodeFree(&cvode_mem);

  return(0);
}

/*
 *-------------------------------
 * Private helper functions
```

```c
 *--------------------------------
 */

/* Allocate memory for data structure of type UserData */

static UserData AllocUserData(void)
{
  int jx, jy;
  UserData data;

  data = (UserData) malloc(sizeof *data);

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      (data->P)[jx][jy] = denalloc(NUM_SPECIES, NUM_SPECIES);
      (data->Jbd)[jx][jy] = denalloc(NUM_SPECIES, NUM_SPECIES);
      (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
    }
  }

  return(data);
}

/* Load problem constants in data */

static void InitUserData(UserData data)
{
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/(MX-1);
  data->dy = (YMAX-YMIN)/(MY-1);
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(TWO*data->dx);
  data->vdco = (ONE/SQR(data->dy))*KV0;
}

/* Free data memory */

static void FreeUserData(UserData data)
{
  int jx, jy;

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      denfree((data->P)[jx][jy]);
      denfree((data->Jbd)[jx][jy]);
      denfreepiv((data->pivot)[jx][jy]);
    }
  }

  free(data);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, realtype dx, realtype dy)
{
  int jx, jy;
  realtype x, y, cx, cy;
  realtype *udata;

  /* Set pointer to data array in vector u. */

  udata = NV_DATA_S(u);

  /* Load initial profiles of c1 and c2 into u vector */

  for (jy=0; jy < MY; jy++) {
    y = YMIN + jy*dy;
    cy = SQR(RCONST(0.1)*(y - YMID));
    cy = ONE - cy + RCONST(0.5)*SQR(cy);
    for (jx=0; jx < MX; jx++) {
      x = XMIN + jx*dx;
      cx = SQR(RCONST(0.1)*(x - XMID));
      cx = ONE - cx + RCONST(0.5)*SQR(cx);
      IJKth(udata,1,jx,jy) = C1_SCALE*cx*cy;
      IJKth(udata,2,jx,jy) = C2_SCALE*cx*cy;
    }
  }
}

/* Print current t, step count, order, stepsize, and sampled c1,c2 values */

static void PrintOutput(void *cvode_mem, N_Vector u, realtype t)
{
  long int nst;
  int qu, flag;
  realtype hu, *udata;
  int mxh = MX/2 - 1, myh = MY/2 - 1, mx1 = MX - 1, my1 = MY - 1;

  udata = NV_DATA_S(u);

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetLastOrder(cvode_mem, &qu);
  check_flag(&flag, "CVodeGetLastOrder", 1);
  flag = CVodeGetLastStep(cvode_mem, &hu);
  check_flag(&flag, "CVodeGetLastStep", 1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("t␣=␣%.2Le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2Le\n",
         t, nst, qu, hu);
  printf("c1␣(bot.left/middle/top␣rt.)␣=␣%12.3Le␣␣%12.3Le␣␣%12.3Le\n",
         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
  printf("c2␣(bot.left/middle/top␣rt.)␣=␣%12.3Le␣␣%12.3Le␣␣%12.3Le\n\n",
         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("t␣=␣%.2le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2le\n",
         t, nst, qu, hu);
  printf("c1␣(bot.left/middle/top␣rt.)␣=␣%12.3le␣␣%12.3le␣␣%12.3le\n",
         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
  printf("c2␣(bot.left/middle/top␣rt.)␣=␣%12.3le␣␣%12.3le␣␣%12.3le\n\n",
         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
#else
  printf("t␣=␣%.2e␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2e\n",
         t, nst, qu, hu);
  printf("c1␣(bot.left/middle/top␣rt.)␣=␣%12.3e␣␣%12.3e␣␣%12.3e\n",
         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
  printf("c2␣(bot.left/middle/top␣rt.)␣=␣%12.3e␣␣%12.3e␣␣%12.3e\n\n",
         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
#endif
}

/* Get and print final statistics */

static void PrintFinalStats(void *cvode_mem)
{
  long int lenrw, leniw ;
  long int lenrwLS, leniwLS;
  long int nst, nfe, nsetups, nni, ncfn, netf;
  long int nli, npe, nps, ncfl, nfeLS;
  int flag;

  flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
  check_flag(&flag, "CVodeGetWorkSpace", 1);
  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);

  flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
  check_flag(&flag, "CVSpilsGetWorkSpace", 1);
  flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
  check_flag(&flag, "CVSpilsGetNumLinIters", 1);
  flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
  check_flag(&flag, "CVSpilsGetNumPrecEvals", 1);
  flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
  check_flag(&flag, "CVSpilsGetNumPrecSolves", 1);
  flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
  check_flag(&flag, "CVSpilsGetNumConvFails", 1);
  flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
  check_flag(&flag, "CVSpilsGetNumRhsEvals", 1);

  printf("\nFinal␣Statistics..␣\n\n");
  printf("lenrw␣␣␣=␣%5ld␣␣␣␣␣leniw␣␣␣=␣%5ld\n", lenrw, leniw);
  printf("lenrwLS␣=␣%5ld␣␣␣␣␣leniwLS␣=␣%5ld\n", lenrwLS, leniwLS);
  printf("nst␣␣␣␣␣=␣%5ld\n"                    , nst);
  printf("nfe␣␣␣␣␣=␣%5ld␣␣␣␣␣nfeLS␣␣␣=␣%5ld\n"  , nfe, nfeLS);
  printf("nni␣␣␣␣␣=␣%5ld␣␣␣␣␣nli␣␣␣␣␣=␣%5ld\n"  , nni, nli);
  printf("nsetups␣=␣%5ld␣␣␣␣␣netf␣␣␣␣=␣%5ld\n"  , nsetups, netf);
  printf("npe␣␣␣␣␣=␣%5ld␣␣␣␣␣nps␣␣␣␣␣=␣%5ld\n"  , npe, nps);
  printf("ncfn␣␣␣␣=␣%5ld␣␣␣␣␣ncfl␣␣␣␣=␣%5ld\n\n", ncfn, ncfl);
}

/* Check function return value...
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns a flag so check if
              flag >= 0
     opt == 2 means function allocates memory so check if returned
```

```
                NULL pointer */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1); }

  return(0);
}

/*
 *-------------------------------
 * Functions called by the solver
 *-------------------------------
 */

/* f routine. Compute RHS function f(t,u). */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
  realtype q4coef, dely, verdco, hordco, horaco;
  realtype *udata, *dudata;
  int jx, jy, idn, iup, ileft, iright;
  UserData data;

  data = (UserData) f_data;
  udata = NV_DATA_S(u);
  dudata = NV_DATA_S(udot);

  /* Set diurnal rate coefficients. */

  s = sin(data->om*t);
  if (s > ZERO) {
    q3 = EXP(-A3/s);
    data->q4 = EXP(-A4/s);
  } else {
      q3 = ZERO;
      data->q4 = ZERO;
  }

  /* Make local copies of problem variables, for efficiency. */

  q4coef = data->q4;
  dely = data->dy;
  verdco = data->vdco;
  hordco  = data->hdco;
  horaco  = data->haco;

  /* Loop over all grid points. */

  for (jy=0; jy < MY; jy++) {

    /* Set vertical diffusion coefficients at jy +- 1/2 */

    ydn = YMIN + (jy - RCONST(0.5))*dely;
    yup = ydn + dely;
    cydn = verdco*EXP(RCONST(0.2)*ydn);
    cyup = verdco*EXP(RCONST(0.2)*yup);
    idn = (jy == 0) ? 1 : -1;
    iup = (jy == MY-1) ? -1 : 1;
    for (jx=0; jx < MX; jx++) {

      /* Extract c1 and c2, and set kinetic rate terms. */

      c1 = IJKth(udata,1,jx,jy);
      c2 = IJKth(udata,2,jx,jy);
      qq1 = Q1*c1*C3;
      qq2 = Q2*c1*c2;
      qq3 = q3*C3;
      qq4 = q4coef*c2;
      rkin1 = -qq1 - qq2 + TWO*qq3 + qq4;
      rkin2 = qq1 - qq2 - qq4;

      /* Set vertical diffusion terms. */

      c1dn = IJKth(udata,1,jx,jy+idn);
      c2dn = IJKth(udata,2,jx,jy+idn);
      c1up = IJKth(udata,1,jx,jy+iup);
      c2up = IJKth(udata,2,jx,jy+iup);
      vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
      vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);

      /* Set horizontal diffusion and advection terms. */

      ileft = (jx == 0) ? 1 : -1;
      iright =(jx == MX-1) ? -1 : 1;
      c1lt = IJKth(udata,1,jx+ileft,jy);
      c2lt = IJKth(udata,2,jx+ileft,jy);
      c1rt = IJKth(udata,1,jx+iright,jy);
      c2rt = IJKth(udata,2,jx+iright,jy);
      hord1 = hordco*(c1rt - TWO*c1 + c1lt);
      hord2 = hordco*(c2rt - TWO*c2 + c2lt);
      horad1 = horaco*(c1rt - c1lt);
      horad2 = horaco*(c2rt - c2lt);

      /* Load all terms into udot. */

      IJKth(dudata, 1, jx, jy) = vertd1 + hord1 + horad1 + rkin1;
      IJKth(dudata, 2, jx, jy) = vertd2 + hord2 + horad2 + rkin2;
    }
  }

  return(0);
}

/* Preconditioner setup routine. Generate and preprocess P. */

static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr, realtype gamma,
                   void *P_data, N_Vector vtemp1, N_Vector vtemp2,
                   N_Vector vtemp3)
{
  realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
  realtype **(*P)[MY], **(*Jbd)[MY];
  long int *(*pivot)[MY], ier;
  int jx, jy;
  realtype *udata, **a, **j;
  UserData data;

  /* Make local copies of pointers in P_data, and of pointer to u's data */

  data = (UserData) P_data;
  P = data->P;
  Jbd = data->Jbd;
  pivot = data->pivot;
  udata = NV_DATA_S(u);

  if (jok) {

    /* jok = TRUE: Copy Jbd to P */

    for (jy=0; jy < MY; jy++)
      for (jx=0; jx < MX; jx++)
        dencopy(Jbd[jx][jy], P[jx][jy], NUM_SPECIES, NUM_SPECIES);

    *jcurPtr = FALSE;

  }

  else {
    /* jok = FALSE: Generate Jbd from scratch and copy to P */

    /* Make local copies of problem variables, for efficiency. */

    q4coef = data->q4;
    dely = data->dy;
    verdco = data->vdco;
    hordco  = data->hdco;

    /* Compute 2x2 diagonal Jacobian blocks (using q4 values
       computed on the last f call).  Load into P. */

    for (jy=0; jy < MY; jy++) {
      ydn = YMIN + (jy - RCONST(0.5))*dely;
      yup = ydn + dely;
      cydn = verdco*EXP(RCONST(0.2)*ydn);
      cyup = verdco*EXP(RCONST(0.2)*yup);
      diag = -(cydn + cyup + TWO*hordco);
      for (jx=0; jx < MX; jx++) {
        c1 = IJKth(udata,1,jx,jy);
        c2 = IJKth(udata,2,jx,jy);
        j = Jbd[jx][jy];
        a = P[jx][jy];
        IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
        IJth(j,1,2) = -Q2*c1 + q4coef;
        IJth(j,2,1) = Q1*C3 - Q2*c2;
        IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
        dencopy(j, a, NUM_SPECIES, NUM_SPECIES);
      }
    }

    *jcurPtr = TRUE;

  }

  /* Scale by -gamma */

  for (jy=0; jy < MY; jy++)
    for (jx=0; jx < MX; jx++)
      denscale(-gamma, P[jx][jy], NUM_SPECIES, NUM_SPECIES);

  /* Add identity matrix and do LU decompositions on blocks in place. */

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      denaddI(P[jx][jy], NUM_SPECIES);
      ier = denGETRF(P[jx][jy], NUM_SPECIES, NUM_SPECIES, pivot[jx][jy]);
      if (ier != 0) return(1);
    }
  }

  return(0);
}

/* Preconditioner solve routine */

static int PSolve(realtype tn, N_Vector u, N_Vector fu,
                  N_Vector r, N_Vector z,
                  realtype gamma, realtype delta,
                  int lr, void *P_data, N_Vector vtemp)
{
  realtype **(*P)[MY];
  long int *(*pivot)[MY];
  int jx, jy;
  realtype *zdata, *v;
  UserData data;

  /* Extract the P and pivot arrays from P_data. */

  data = (UserData) P_data;
  P = data->P;
  pivot = data->pivot;
  zdata = NV_DATA_S(z);

  N_VScale(ONE, r, z);

  /* Solve the block-diagonal system Px = r using LU factors stored
     in P and pivot data in pivot, and return the solution in z. */

  for (jx=0; jx < MX; jx++) {
    for (jy=0; jy < MY; jy++) {
      v = &(IJKth(zdata, 1, jx, jy));
      denGETRS(P[jx][jy], NUM_SPECIES, pivot[jx][jy], v);
    }
  }

  return(0);
}
```

D Listing of cvnonx_p.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.1 $
 * $Date: 2006/07/05 15:50:05 $
 * -----------------------------------------------------------------
 * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, George Byrne,
 *                and Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * The following is a simple example problem, with the program for
 * its solution by CVODE. The problem is the semi-discrete
 * form of the advection-diffusion equation in 1-D:
 *   du/dt = d^2 u / dx^2 + .5 du/dx
 * on the interval 0 <= x <= 2, and the time interval 0 <= t <= 5.
 * Homogeneous Dirichlet boundary conditions are posed, and the
 * initial condition is the following:
 *   u(x,t=0) = x(2-x)exp(2x) .
 * The PDE is discretized on a uniform grid of size MX+2 with
 * central differencing, and with boundary values eliminated,
 * leaving an ODE system of size NEQ = MX.
 * This program solves the problem with the option for nonstiff
 * systems: ADAMS method and functional iteration.
 * It uses scalar relative and absolute tolerances.
 * Output is printed at t = .5, 1.0, ..., 5.
 * Run statistics (optional outputs) are printed at the end.
 *
 * This version uses MPI for user routines.
 * Execute with Number of Processors = N,  with 1 <= N <= MX.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <cvode/cvode.h>                /* prototypes for CVODE fcts. */
#include <nvector/nvector_parallel.h> /* definition of N_Vector and macros */
#include <sundials/sundials_types.h>  /* definition of realtype */
#include <sundials/sundials_math.h>   /* definition of EXP */

#include <mpi.h>                        /* MPI constants and types */

/* Problem Constants */

#define ZERO  RCONST(0.0)

#define XMAX  RCONST(2.0)    /* domain boundary           */
#define MX    10             /* mesh dimension            */
#define NEQ   MX             /* number of equations       */
#define ATOL  RCONST(1.0e-5) /* scalar absolute tolerance */
#define T0    ZERO           /* initial time              */
#define T1    RCONST(0.5)    /* first output time         */
#define DTOUT RCONST(0.5)    /* output time increment     */
#define NOUT  10             /* number of output times    */

/* Type : UserData
```

```
   contains grid constants, parallel machine parameters, work array. */

typedef struct {
  realtype dx, hdcoef, hacoef;
  int npes, my_pe;
  MPI_Comm comm;
  realtype z[100];
} *UserData;

/* Private Helper Functions */

static void SetIC(N_Vector u, realtype dx, long int my_length,
                  long int my_base);

static void PrintIntro(int npes);

static void PrintData(realtype t, realtype umax, long int nst);

static void PrintFinalStats(void *cvode_mem);

/* Functions Called by the Solver */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);

/* Private function to check function return values */

static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/***************************** Main Program ******************************/

int main(int argc, char *argv[])
{
  realtype dx, reltol, abstol, t, tout, umax;
  N_Vector u;
  UserData data;
  void *cvode_mem;
  int iout, flag, my_pe, npes;
  long int local_N, nperpe, nrem, my_base, nst;

  MPI_Comm comm;

  u = NULL;
  data = NULL;
  cvode_mem = NULL;

  /* Get processor number, total number of pe's, and my_pe. */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  /* Set local vector length. */
  nperpe = NEQ/npes;
  nrem = NEQ - npes*nperpe;
  local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
  my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;

  data = (UserData) malloc(sizeof *data);  /* Allocate data memory */
  if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
```

```c

  data->comm = comm;
  data->npes = npes;
  data->my_pe = my_pe;

  u = N_VNew_Parallel(comm, local_N, NEQ);  /* Allocate u vector */
  if(check_flag((void *)u, "N_VNew", 0, my_pe)) MPI_Abort(comm, 1);

  reltol = ZERO;  /* Set the tolerances */
  abstol = ATOL;

  dx = data->dx = XMAX/((realtype)(MX+1));  /* Set grid coefficients in data */
  data->hdcoef = RCONST(1.0)/(dx*dx);
  data->hacoef = RCONST(0.5)/(RCONST(2.0)*dx);

  SetIC(u, dx, local_N, my_base);  /* Initialize u vector */

  /*
     Call CVodeCreate to create the solver memory:

     CV_ADAMS   specifies the Adams Method
     CV_FUNCTIONAL  specifies functional iteration

     A pointer to the integrator memory is returned and stored in cvode_mem.
  */

  cvode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
  if(check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);

  flag = CVodeSetFdata(cvode_mem, data);
  if(check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  /*
     Call CVodeMalloc to initialize the integrator memory:

     cvode_mem is the pointer to the integrator memory returned by CVodeCreate
     f       is the user's right hand side function in y'=f(t,y)
     T0      is the initial time
     u       is the initial dependent variable vector
     CV_SS   specifies scalar relative and absolute tolerances
     reltol  is the relative tolerance
     &abstol is a pointer to the scalar absolute tolerance
  */

  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  if (my_pe == 0) PrintIntro(npes);

  umax = N_VMaxNorm(u);

  if (my_pe == 0) {
    t = T0;
    PrintData(t, umax, 0);
  }

  /* In loop over output points, call CVode, print results, test for error */

  for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    if(check_flag(&flag, "CVode", 1, my_pe)) break;
    umax = N_VMaxNorm(u);
    flag = CVodeGetNumSteps(cvode_mem, &nst);
    check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
    if (my_pe == 0) PrintData(t, umax, nst);
  }

  if (my_pe == 0)
    PrintFinalStats(cvode_mem);  /* Print some final statistics */

  N_VDestroy_Parallel(u);        /* Free the u vector */
  CVodeFree(&cvode_mem);         /* Free the integrator memory */
  free(data);                    /* Free user data */

  MPI_Finalize();

  return(0);
}

/*********************** Private Helper Functions **********************/

/* Set initial conditions in u vector */

static void SetIC(N_Vector u, realtype dx, long int my_length,
                  long int my_base)
{
  int i;
  long int iglobal;
  realtype x;
  realtype *udata;

  /* Set pointer to data array and get local length of u. */
  udata = NV_DATA_P(u);
  my_length = NV_LOCLENGTH_P(u);

  /* Load initial profile into u vector */
  for (i=1; i<=my_length; i++) {
    iglobal = my_base + i;
    x = iglobal*dx;
    udata[i-1] = x*(XMAX - x)*EXP(RCONST(2.0)*x);
  }
}

/* Print problem introduction */

static void PrintIntro(int npes)
{
  printf("\n␣1-D␣advection-diffusion␣equation,␣mesh␣size␣=%3d␣\n", MX);
  printf("\n␣Number␣of␣PEs␣=␣%3d␣\n\n", npes);

  return;
}

/* Print data */

static void PrintData(realtype t, realtype umax, long int nst)
{

```

```
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("At␣t␣=␣%4.2Lf␣␣max.norm(u)␣=%14.6Le␣␣nst␣=%4ld␣\n", t, umax, nst);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("At␣t␣=␣%4.2f␣␣max.norm(u)␣=%14.6le␣␣nst␣=%4ld␣\n", t, umax, nst);
#else
  printf("At␣t␣=␣%4.2f␣␣max.norm(u)␣=%14.6e␣␣nst␣=%4ld␣\n", t, umax, nst);
#endif

  return;
}

/* Print some final statistics located in the iopt array */

static void PrintFinalStats(void *cvode_mem)
{
  long int nst, nfe, nni, ncfn, netf;
  int flag;

  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1, 0);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);

  printf("\nFinal␣Statistics:␣\n\n");
  printf("nst␣=␣%-6ld␣␣nfe␣␣=␣%-6ld␣␣", nst, nfe);
  printf("nni␣=␣%-6ld␣␣ncfn␣=␣%-6ld␣␣netf␣=␣%ld\n␣\n", nni, ncfn, netf);
}

/***************** Function Called by the Solver ***********************/

/* f routine. Compute f(t,u). */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype ui, ult, urt, hordc, horac, hdiff, hadv;
  realtype *udata, *dudata, *z;
  int i;
  int npes, my_pe, my_length, my_pe_m1, my_pe_p1, last_pe, my_last;
  UserData data;
  MPI_Status status;
  MPI_Comm comm;

  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);

  /* Extract needed problem constants from data */
  data = (UserData) f_data;
  hordc = data->hdcoef;
  horac = data->hacoef;

  /* Extract parameters for parallel computation. */
  comm = data->comm;
  npes = data->npes;           /* Number of processes. */
  my_pe = data->my_pe;          /* Current process number. */
  my_length = NV_LOCLENGTH_P(u); /* Number of local elements of u. */
  z = data->z;

  /* Compute related parameters. */
  my_pe_m1 = my_pe - 1;
  my_pe_p1 = my_pe + 1;
  last_pe = npes - 1;
  my_last = my_length - 1;

  /* Store local segment of u in the working array z. */
   for (i = 1; i <= my_length; i++)
     z[i] = udata[i - 1];

  /* Pass needed data to processes before and after current process. */
   if (my_pe != 0)
     MPI_Send(&z[1], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
   if (my_pe != last_pe)
     MPI_Send(&z[my_length], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);

  /* Receive needed data from processes before and after current process. */
   if (my_pe != 0)
     MPI_Recv(&z[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
   else z[0] = ZERO;
   if (my_pe != last_pe)
     MPI_Recv(&z[my_length+1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
              &status);
   else z[my_length + 1] = ZERO;

  /* Loop over all grid points in current process. */
  for (i=1; i<=my_length; i++) {

    /* Extract u at x_i and two neighboring points */
    ui = z[i];
    ult = z[i-1];
    urt = z[i+1];

    /* Set diffusion and advection terms and load into udot */
    hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
    hadv = horac*(urt - ult);
    dudata[i-1] = hdiff + hadv;
  }

  return(0);
}

/* Check function return value...
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns a flag so check if
              flag >= 0
     opt == 2 means function allocates memory so check if returned
              NULL pointer */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1); }

  return(0);
}
```

E Listing of cvkryx_p.c

```c
/*
 * -----------------------------------------------------------------
 * $Revision: 1.2 $
 * $Date: 2006/10/11 16:33:55 $
 * -----------------------------------------------------------------
 * Programmer(s): S. D. Cohen, A. C. Hindmarsh, M. R. Wittman, and
 *                Radu Serban  @ LLNL
 * -----------------------------------------------------------------
 * Example problem:
 *
 * An ODE system is generated from the following 2-species diurnal
 * kinetics advection-diffusion PDE system in 2 space dimensions:
 *
 * dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)
 *                 + Ri(c1,c2,t)      for i = 1,2,   where
 *   R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,
 *   R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,
 *   Kv(y) = Kv0*exp(y/5) ,
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)
 * vary diurnally. The problem is posed on the square
 *   0 <= x <= 20,    30 <= y <= 50   (all in km),
 * with homogeneous Neumann boundary conditions, and for time t in
 *   0 <= t <= 86400 sec (1 day).
 * The PDE system is treated by central differences on a uniform
 * mesh, with simple polynomial initial profiles.
 *
 * The problem is solved by CVODE on NPE processors, treated
 * as a rectangular process grid of size NPEX by NPEY, with
 * NPE = NPEX*NPEY. Each processor contains a subgrid of size MXSUB
 * by MYSUB of the (x,y) mesh.  Thus the actual mesh sizes are
 * MX = MXSUB*NPEX and MY = MYSUB*NPEY, and the ODE system size is
 * neq = 2*MX*MY.
 *
 * The solution is done with the BDF/GMRES method (i.e. using the
 * CVSPGMR linear solver) 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 routine.
 *
 * Performance data and sampled solution values are printed at
 * selected output times, and all performance counters are printed
 * on completion.
 *
 * This version uses MPI for user routines.
 *
 * Execution: mpirun -np N cvkryx_p   with N = NPEX*NPEY (see
 * constants below).
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <cvode/cvode.h>                /* prototypes for CVODE fcts. */
#include <cvode/cvode_spgmr.h>          /* prototypes and constants for CVSPGMR solver */
```

```
#include <nvector/nvector_parallel.h>     /* definition N_Vector and macro NV_DATA_P
*/
#include <sundials/sundials_smalldense.h> /* prototypes for small dense matrix fcts. */
#include <sundials/sundials_types.h>      /* definitions of realtype, booleantype */
#include <sundials/sundials_math.h>       /* definition of macros SQR and EXP */

#include <mpi.h>                          /* MPI constants and types */

/* Problem Constants */

#define NVARS        2                    /* number of species          */
#define KH           RCONST(4.0e-6)       /* horizontal diffusivity Kh */
#define VEL          RCONST(0.001)        /* advection velocity V       */
#define KV0          RCONST(1.0e-8)       /* coefficient in Kv(y)       */
#define Q1           RCONST(1.63e-16)     /* coefficients q1, q2, c3    */
#define Q2           RCONST(4.66e-16)
#define C3           RCONST(3.7e16)
#define A3           RCONST(22.62)     /* coefficient in expression for q3(t) */
#define A4           RCONST(7.601)     /* coefficient in expression for q4(t) */
#define C1_SCALE     RCONST(1.0e6)     /* coefficients in initial profiles
*/
#define C2_SCALE     RCONST(1.0e12)

#define T0           RCONST(0.0)          /* initial time */
#define NOUT         12                   /* number of output times */
#define TWOHR        RCONST(7200.0)       /* number of seconds in two hours  */
#define HALFDAY      RCONST(4.32e4)       /* number of seconds in a half day */
#define PI       RCONST(3.1415926535898)  /* pi */

#define XMIN         RCONST(0.0)          /* grid boundaries in x  */
#define XMAX         RCONST(20.0)
#define YMIN         RCONST(30.0)         /* grid boundaries in y  */
#define YMAX         RCONST(50.0)

#define NPEX         2              /* no. PEs in x direction of PE array */
#define NPEY         2              /* no. PEs in y direction of PE array */
                                    /* Total no. PEs = NPEX*NPEY */
#define MXSUB        5              /* no. x points per subgrid */
#define MYSUB        5              /* no. y points per subgrid */

#define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
#define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
                                    /* Spatial mesh is MX by MY */
/* CVodeMalloc Constants */

#define RTOL    RCONST(1.0e-5)    /* scalar relative tolerance */
#define FLOOR   RCONST(100.0)     /* value of C1 or C2 at which tolerances */
                                  /* change from relative to absolute      */
#define ATOL    (RTOL*FLOOR)      /* scalar absolute tolerance */


/* User-defined matrix accessor macro: IJth */

/* IJth is defined in order to write code which indexes into small dense
   matrices with a (row,column) pair, where 1 <= row,column <= NVARS.

   IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
   where 1 <= i,j <= NVARS. The small matrix routines in dense.h
   work with matrices stored by column in a 2-dimensional array. In C,
   arrays are indexed starting at 0, not 1. */

#define IJth(a,i,j) (a[j-1][i-1])

/* Type : UserData
   contains problem constants, preconditioner blocks, pivot arrays,
   grid constants, and processor indices */

typedef struct {
  realtype q4, om, dx, dy, hdco, haco, vdco;
  realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
  int my_pe, isubx, isuby;
  long int nvmxsub, nvmxsub2;
  MPI_Comm comm;
} *UserData;

typedef struct {
  void *f_data;
  realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
  long int *pivot[MXSUB][MYSUB];
} *PreconData;


/* Private Helper Functions */

static PreconData AllocPreconData(UserData data);
static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
static void FreePreconData(PreconData pdata);
static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        N_Vector u, realtype t);
static void PrintFinalStats(void *cvode_mem);
static void BSend(MPI_Comm comm,
                  int my_pe, int isubx, int isuby,
                  long int dsizex, long int dsizey,
                  realtype udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[],
                      int my_pe, int isubx, int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[]);
static void BRecvWait(MPI_Request request[],
                      int isubx, int isuby,
                      long int dsizex, realtype uext[],
                      realtype buffer[]);
static void ucomm(realtype t, N_Vector u, UserData data);
static void fcalc(realtype t, realtype udata[], realtype dudata[],
                  UserData data);


/* Functions Called by the Solver */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);

static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr,
                   realtype gamma, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

static int PSolve(realtype tn, N_Vector u, N_Vector fu,
                    N_Vector r, N_Vector z,
                    realtype gamma, realtype delta,
                    int lr, void *P_data, N_Vector vtemp);


/* Private function to check function return values */

static int check_flag(void *flagvalue, char *funcname, int opt, int id);


/***************************** Main Program ******************************/

int main(int argc, char *argv[])
{
  realtype abstol, reltol, t, tout;
  N_Vector u;
  UserData data;
  PreconData predata;
  void *cvode_mem;
  int iout, flag, my_pe, npes;
  long int neq, local_N;
  MPI_Comm comm;

  u = NULL;
  data = NULL;
  predata = NULL;
  cvode_mem = NULL;

  /* Set problem size neq */
  neq = NVARS*MX*MY;

  /* Get processor number and total number of pe's */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  if (npes != NPEX*NPEY) {
    if (my_pe == 0)
      fprintf(stderr, "\nMPI_ERROR(0):␣npes␣=␣%d␣is␣not␣equal␣to␣NPEX*NPEY␣=␣%d\n\n",
              npes,NPEX*NPEY);
    MPI_Finalize();
    return(1);
  }

  /* Set local length */
  local_N = NVARS*MXSUB*MYSUB;

  /* Allocate and load user data block; allocate preconditioner block */
  data = (UserData) malloc(sizeof *data);
  if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
  InitUserData(my_pe, comm, data);
  predata = AllocPreconData (data);

  /* Allocate u, and set initial values and tolerances */
  u = N_VNew_Parallel(comm, local_N, neq);
  if (check_flag((void *)u, "N_VNew", 0, my_pe)) MPI_Abort(comm, 1);
  SetInitialProfiles(u, data);
  abstol = ATOL; reltol = RTOL;

  /*
     Call CVodeCreate to create the solver memory:

     CV_BDF     specifies the Backward Differentiation Formula
     CV_NEWTON  specifies a Newton iteration

     A pointer to the integrator memory is returned and stored in cvode_mem.
  */
  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if (check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);

  /* Set the pointer to user-defined data */
  flag = CVodeSetFdata(cvode_mem, data);
  if (check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  /*
     Call CVodeMalloc to initialize the integrator memory:

     cvode_mem is the pointer to the integrator memory returned by CVodeCreate
     f       is the user's right hand side function in y'=f(t,y)
     T0      is the initial time
     u       is the initial dependent variable vector
     CV_SS   specifies scalar relative and absolute tolerances
     reltol  is the relative tolerance
     &abstol is a pointer to the scalar absolute tolerance
  */
  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  /* Call CVSpgmr to specify the linear solver CVSPGMR
     with left preconditioning and the maximum Krylov dimension maxl */
  flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
  if (check_flag(&flag, "CVSpgmr", 1, my_pe)) MPI_Abort(comm, 1);

  /* Set preconditioner setup and solve routines Precond and PSolve,
     and the pointer to the user-defined block data */
  flag = CVSpilsSetPreconditioner(cvode_mem, Precond, PSolve, predata);
  if (check_flag(&flag, "CVSpilsSetPreconditioner", 1, my_pe)) MPI_Abort(comm, 1);

  if (my_pe == 0)
    printf("\n2-species␣diurnal␣advection-diffusion␣problem\n\n");

  /* In loop over output points, call CVode, print results, test for error */
  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    if (check_flag(&flag, "CVode", 1, my_pe)) break;
    PrintOutput(cvode_mem, my_pe, comm, u, t);
  }

  /* Print final statistics */
  if (my_pe == 0) PrintFinalStats(cvode_mem);

  /* Free memory */
  N_VDestroy_Parallel(u);
  free(data);
  FreePreconData(predata);
  CVodeFree(&cvode_mem);

  MPI_Finalize();

  return(0);
}


/*********************** Private Helper Functions ************************/

/* Allocate memory for data structure of type UserData */

static PreconData AllocPreconData(UserData fdata)
{
  int lx, ly;
  PreconData pdata;

  pdata = (PreconData) malloc(sizeof *pdata);

  pdata->f_data = fdata;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      (pdata->P)[lx][ly] = denalloc(NVARS, NVARS);
      (pdata->Jbd)[lx][ly] = denalloc(NVARS, NVARS);
      (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
    }
  }

  return(pdata);
}

/* Load constants in data */

static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
{
  int isubx, isuby;

  /* Set problem constants */
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/((realtype)(MX-1));
  data->dy = (YMAX-YMIN)/((realtype)(MY-1));
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(RCONST(2.0)*data->dx);
  data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;

  /* Set machine-related constants */
  data->comm = comm;
  data->my_pe = my_pe;

  /* isubx and isuby are the PE grid indices corresponding to my_pe */
  isuby = my_pe/NPEX;
  isubx = my_pe - isuby*NPEX;
  data->isubx = isubx;
  data->isuby = isuby;

  /* Set the sizes of a boundary x-line in u and uext */
  data->nvmxsub = NVARS*MXSUB;
  data->nvmxsub2 = NVARS*(MXSUB+2);
}
```

```
/* Free preconditioner data memory */

static void FreePreconData(PreconData pdata)
{
  int lx, ly;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denfree((pdata->P)[lx][ly]);
      denfree((pdata->Jbd)[lx][ly]);
      denfreepiv((pdata->pivot)[lx][ly]);
    }
  }

  free(pdata);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, UserData data)
{
  int isubx, isuby, lx, ly, jx, jy;
  long int offset;
  realtype dx, dy, x, y, cx, cy, xmid, ymid;
  realtype *udata;

  /* Set pointer to data array in vector u */
  udata = NV_DATA_P(u);

  /* Get mesh spacings, and subgrid indices for this PE */
  dx = data->dx;          dy = data->dy;
  isubx = data->isubx;    isuby = data->isuby;

  /* Load initial profiles of c1 and c2 into local u vector.
  Here lx and ly are local mesh point indices on the local subgrid,
  and jx and jy are the global mesh point indices. */
  offset = 0;
  xmid = RCONST(0.5)*(XMIN + XMAX);
  ymid = RCONST(0.5)*(YMIN + YMAX);
  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    y = YMIN + jy*dy;
    cy = SQR(RCONST(0.1)*(y - ymid));
    cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      x = XMIN + jx*dx;
      cx = SQR(RCONST(0.1)*(x - xmid));
      cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
      udata[offset  ] = C1_SCALE*cx*cy;
      udata[offset+1] = C2_SCALE*cx*cy;
      offset = offset + 2;
    }
  }
}

/* Print current t, step count, order, stepsize, and sampled c1,c2 values */

static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        N_Vector u, realtype t)
{
  int qu, flag;
  realtype hu, *udata, tempu[2];
  int npelast;
  long int i0, i1, nst;
  MPI_Status status;

  npelast = NPEX*NPEY - 1;
  udata = NV_DATA_P(u);

  /* Send c1,c2 at top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NVARS*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
      MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      tempu[0] = udata[i0];
      tempu[1] = udata[i1];
    }
  }

  /* On PE 0, receive c1,c2 at top right, then print performance data
     and sampled solution values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    flag = CVodeGetNumSteps(cvode_mem, &nst);
    check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
    flag = CVodeGetLastOrder(cvode_mem, &qu);
    check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
    flag = CVodeGetLastStep(cvode_mem, &hu);
    check_flag(&flag, "CVodeGetLastStep", 1, my_pe);

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("t␣=␣%.2Le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2Le\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3Le␣%12.3Le␣\n", udata[0], udata[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3Le␣%12.3Le␣\n\n", tempu[0], tempu[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("t␣=␣%.2le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2le\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3le␣%12.3le␣\n", udata[0], udata[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3le␣%12.3le␣\n\n", tempu[0], tempu[1]);
#else
    printf("t␣=␣%.2e␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2e\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3e␣%12.3e␣\n", udata[0], udata[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3e␣%12.3e␣\n\n", tempu[0], tempu[1]);
#endif
  }
}

/* Print final statistics contained in iopt */

static void PrintFinalStats(void *cvode_mem)
{
  long int lenrw, leniw ;
  long int lenrwLS, leniwLS;
  long int nst, nfe, nsetups, nni, ncfn, netf;
  long int nli, npe, nps, ncfl, nfeLS;
  int flag;

  flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
  check_flag(&flag, "CVodeGetWorkSpace", 1, 0);
  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1, 0);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);

  flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
  check_flag(&flag, "CVSpilsGetWorkSpace", 1, 0);
  flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
  check_flag(&flag, "CVSpilsGetNumLinIters", 1, 0);
  flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
  check_flag(&flag, "CVSpilsGetNumPrecEvals", 1, 0);
  flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
  check_flag(&flag, "CVSpilsGetNumPrecSolves", 1, 0);
  flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
  check_flag(&flag, "CVSpilsGetNumConvFails", 1, 0);
  flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
  check_flag(&flag, "CVSpilsGetNumRhsEvals", 1, 0);

  printf("\nFinal␣Statistics:␣\n\n");
  printf("lenrw␣␣␣=␣%5ld␣␣␣␣␣leniw␣␣␣=␣%5ld\n", lenrw, leniw);
  printf("lenrwls␣=␣%5ld␣␣␣␣␣leniwls␣=␣%5ld\n", lenrwLS, leniwLS);
  printf("nst␣␣␣␣␣=␣%5ld\n"                   , nst);
  printf("nfe␣␣␣␣␣=␣%5ld␣␣␣␣␣nfels␣␣␣=␣%5ld\n"  , nfe, nfeLS);
  printf("nni␣␣␣␣␣=␣%5ld␣␣␣␣␣nli␣␣␣␣␣=␣%5ld\n"  , nni, nli);
  printf("nsetups␣=␣%5ld␣␣␣␣␣netf␣␣␣␣=␣%5ld\n"  , nsetups, netf);
  printf("npe␣␣␣␣␣=␣%5ld␣␣␣␣␣nps␣␣␣␣␣=␣%5ld\n"  , npe, nps);
  printf("ncfn␣␣␣␣=␣%5ld␣␣␣␣␣ncfl␣␣␣␣=␣%5ld\n\n", ncfn, ncfl);
}

/* Routine to send boundary data to neighboring PEs */

static void BSend(MPI_Comm comm,
                  int my_pe, int isubx, int isuby,
                  long int dsizex, long int dsizey,
                  realtype udata[])
{
  int i, ly;
  long int offsetu, offsetbuf;
  realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];

  /* If isuby > 0, send data from bottom x-line of u */
  if (isuby != 0)
    MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If isuby < NPEY-1, send data from top x-line of u */
  if (isuby != NPEY-1) {
    offsetu = (MYSUB-1)*dsizex;
    MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
  }

  /* If isubx > 0, send data from left y-line of u (via bufleft) */
  if (isubx != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = ly*dsizex;
      for (i = 0; i < NVARS; i++)
        bufleft[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
  if (isubx != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
      for (i = 0; i < NVARS; i++)
        bufright[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }
}

/* Routine to start receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvPost(MPI_Comm comm, MPI_Request request[],
                      int my_pe, int isubx, int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[])
{
  long int offsetue;
  /* Have bufleft and bufright use the same buffer */
  realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
                                             my_pe-NPEX, 0, comm, &request[0]);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1) {
    offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
                                             my_pe+NPEX, 0, comm, &request[1]);
  }

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                          my_pe-1, 0, comm, &request[2]);
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                          my_pe+1, 0, comm, &request[3]);
  }
}

/* Routine to finish receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvWait(MPI_Request request[],
                      int isubx, int isuby,
                      long int dsizex, realtype uext[],
                      realtype buffer[])
{
  int i, ly;
  long int dsizex2, offsetue, offsetbuf;
  realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NVARS;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Wait(&request[0],&status);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+1)*dsizex2;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufleft[offsetbuf+i];
    }
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Wait(&request[3],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+2)*dsizex2 - NVARS;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufright[offsetbuf+i];
    }
  }
}

/* ucomm routine.  This routine performs all communication
   between processors of data needed to calculate f. */

static void ucomm(realtype t, N_Vector u, UserData data)
{

  realtype *udata, *uext, buffer[2*NVARS*MYSUB];
  MPI_Comm comm;
  int my_pe, isubx, isuby;
  long int nvmxsub, nvmysub;
  MPI_Request request[4];

  udata = NV_DATA_P(u);

  /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
  comm = data->comm;  my_pe = data->my_pe;
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;
  nvmysub = NVARS*MYSUB;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs */
  BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs */
  BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);

  /* Finish receiving boundary data from neighboring PEs */
  BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
}

/* fcalc routine. Compute f(t,y).  This routine assumes that communication
   between processors of data needed to calculate f has already been done,
   and this data is in the work array uext. */

static void fcalc(realtype t, realtype udata[],
                  realtype dudata[], UserData data)
{
  realtype *uext;
  realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
  realtype q4coef, dely, verdco, hordco, horaco;
  int i, lx, ly, jx, jy;
  int isubx, isuby;
  long int nvmxsub, nvmxsub2, offsetu, offsetue;

  /* Get subgrid indices, data sizes, extended work array uext */
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
  uext = data->uext;

  /* Copy local segment of u vector into the working extended array uext */
  offsetu = 0;
  offsetue = nvmxsub2 + NVARS;
  for (ly = 0; ly < MYSUB; ly++) {
    for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
    offsetu = offsetu + nvmxsub;
    offsetue = offsetue + nvmxsub2;
  }

  /* To facilitate homogeneous Neumann boundary conditions, when this is
  a boundary PE, copy data from the first interior mesh line of u to uext */

  /* If isuby = 0, copy x-line 2 of u to uext */
  if (isuby == 0) {
    for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = udata[nvmxsub+i];
  }

  /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
  if (isuby == NPEY-1) {
    offsetu = (MYSUB-2)*nvmxsub;
    offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
    for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
  }

  /* If isubx = 0, copy y-line 2 of u to uext */
  if (isubx == 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*nvmxsub + NVARS;
      offsetue = (ly+1)*nvmxsub2;
      for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
    }
  }

  /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
  if (isubx == NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = (ly+1)*nvmxsub - 2*NVARS;
      offsetue = (ly+2)*nvmxsub2 - NVARS;
      for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
    }
  }

  /* Make local copies of problem variables, for efficiency */
  dely = data->dy;
  verdco = data->vdco;
  hordco  = data->hdco;
  horaco  = data->haco;

  /* Set diurnal rate coefficients as functions of t, and save q4 in
  data block for use by preconditioner evaluation routine */
  s = sin((data->om)*t);
  if (s > RCONST(0.0)) {
    q3 = EXP(-A3/s);
    q4coef = EXP(-A4/s);
  } else {
    q3 = RCONST(0.0);
    q4coef = RCONST(0.0);
  }
  data->q4 = q4coef;

  /* Loop over all grid points in local subgrid */
  for (ly = 0; ly < MYSUB; ly++) {

    jy = ly + isuby*MYSUB;

    /* Set vertical diffusion coefficients at jy +- 1/2 */
    ydn = YMIN + (jy - RCONST(0.5))*dely;
    yup = ydn + dely;
    cydn = verdco*EXP(RCONST(0.2)*ydn);
    cyup = verdco*EXP(RCONST(0.2)*yup);
    for (lx = 0; lx < MXSUB; lx++) {

      jx = lx + isubx*MXSUB;

      /* Extract c1 and c2, and set kinetic rate terms */
      offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
      c1 = uext[offsetue];
      c2 = uext[offsetue+1];
      qq1 = Q1*c1*C3;
      qq2 = Q2*c1*c2;
      qq3 = q3*C3;
      qq4 = q4coef*c2;
      rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
      rkin2 = qq1 - qq2 - qq4;

      /* Set vertical diffusion terms */
      c1dn = uext[offsetue-nvmxsub2];
      c2dn = uext[offsetue-nvmxsub2+1];
      c1up = uext[offsetue+nvmxsub2];
      c2up = uext[offsetue+nvmxsub2+1];
      vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
      vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);

      /* Set horizontal diffusion and advection terms */
      c1lt = uext[offsetue-2];
      c2lt = uext[offsetue-1];
      c1rt = uext[offsetue+2];
      c2rt = uext[offsetue+3];
      hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
      hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
      horad1 = horaco*(c1rt - c1lt);
      horad2 = horaco*(c2rt - c2lt);

      /* Load all terms into dudata */
      offsetu = lx*NVARS + ly*nvmxsub;
      dudata[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
      dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
    }
  }
}


/***************** Functions Called by the Solver *************************/

/* f routine.  Evaluate f(t,y).  First call ucomm to do communication of
   subgrid boundary data into uext.  Then calculate f by a call to fcalc. */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  realtype *udata, *dudata;
  UserData data;

  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);
  data = (UserData) f_data;

  /* Call ucomm to do inter-processor communication */
  ucomm(t, u, data);

  /* Call fcalc to calculate all right-hand sides */
  fcalc(t, udata, dudata, data);

  return(0);
}

/* Preconditioner setup routine. Generate and preprocess P. */
static int Precond(realtype tn, N_Vector u, N_Vector fu,
                   booleantype jok, booleantype *jcurPtr,
                   realtype gamma, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
  realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
  long int nvmxsub, *(*pivot)[MYSUB], ier, offset;
  int lx, ly, jx, jy, isubx, isuby;
  realtype *udata, **a, **j;
  PreconData predata;
  UserData data;

  /* Make local copies of pointers in P_data, pointer to u's data,
     and PE index pair */
  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  Jbd = predata->Jbd;
  pivot = predata->pivot;
  udata = NV_DATA_P(u);
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;

  if (jok) {

  /* jok = TRUE: Copy Jbd to P */
    for (ly = 0; ly < MYSUB; ly++)
      for (lx = 0; lx < MXSUB; lx++)
        dencopy(Jbd[lx][ly], P[lx][ly], NVARS, NVARS);

  *jcurPtr = FALSE;

  }

  else {

  /* jok = FALSE: Generate Jbd from scratch and copy to P */

  /* Make local copies of problem variables, for efficiency */
  q4coef = data->q4;
  dely = data->dy;
  verdco = data->vdco;
  hordco  = data->hdco;

  /* Compute 2x2 diagonal Jacobian blocks (using q4 values
     computed on the last f call).  Load into P. */
    for (ly = 0; ly < MYSUB; ly++) {
      jy = ly + isuby*MYSUB;
      ydn = YMIN + (jy - RCONST(0.5))*dely;
      yup = ydn + dely;
      cydn = verdco*EXP(RCONST(0.2)*ydn);
      cyup = verdco*EXP(RCONST(0.2)*yup);
      diag = -(cydn + cyup + RCONST(2.0)*hordco);
      for (lx = 0; lx < MXSUB; lx++) {
        jx = lx + isubx*MXSUB;
        offset = lx*NVARS + ly*nvmxsub;
        c1 = udata[offset];
        c2 = udata[offset+1];
        j = Jbd[lx][ly];
        a = P[lx][ly];
        IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
        IJth(j,1,2) = -Q2*c1 + q4coef;
        IJth(j,2,1) = Q1*C3 - Q2*c2;
        IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
        dencopy(j, a, NVARS, NVARS);
      }
    }

  *jcurPtr = TRUE;

  }

  /* Scale by -gamma */
    for (ly = 0; ly < MYSUB; ly++)
      for (lx = 0; lx < MXSUB; lx++)
        denscale(-gamma, P[lx][ly], NVARS, NVARS);

  /* Add identity matrix and do LU decompositions on blocks in place */
  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denaddI(P[lx][ly], NVARS);
      ier = denGETRF(P[lx][ly], NVARS, NVARS, pivot[lx][ly]);
      if (ier != 0) return(1);
    }
  }

  return(0);
}

/* Preconditioner solve routine */
static int PSolve(realtype tn, N_Vector u, N_Vector fu,
                  N_Vector r, N_Vector z,
                  realtype gamma, realtype delta,
                  int lr, void *P_data, N_Vector vtemp)
{
  realtype **(*P)[MYSUB];
  long int nvmxsub, *(*pivot)[MYSUB];
  int lx, ly;
  realtype *zdata, *v;
  PreconData predata;
  UserData data;

  /* Extract the P and pivot arrays from P_data */
  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  pivot = predata->pivot;

  /* Solve the block-diagonal system Px = r using LU factors stored
     in P and pivot data in pivot, and return the solution in z.
     First copy vector r to z. */
  N_VScale(RCONST(1.0), r, z);

  nvmxsub = data->nvmxsub;
  zdata = NV_DATA_P(z);

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      v = &(zdata[lx*NVARS + ly*nvmxsub]);
      denGETRS(P[lx][ly], NVARS, pivot[lx][ly], v);
    }
  }

  return(0);
}


/*********************** Private Helper Function ************************/

/* Check function return value...
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns a flag so check if
              flag >= 0
     opt == 2 means function allocates memory so check if returned
              NULL pointer */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
            id, funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣with␣flag␣=␣%d\n\n",
              id, funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
            id, funcname);
      return(1); }

    return(0);
}
```

F Listing of cvkryx_bbd_p.c

```c
/*
 * -----------------------------------------------------------------
 * $Revision: 1.1 $
 * $Date: 2006/07/05 15:50:05 $
 * -----------------------------------------------------------------
 * Programmer(s): S. D. Cohen, A. C. Hindmarsh, M. R. Wittman, and
 *                Radu Serban  @ LLNL
 * --------------------------------------------------------------------
 * Example problem:
 *
 * An ODE system is generated from the following 2-species diurnal
 * kinetics advection-diffusion PDE system in 2 space dimensions:
 *
 * dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)
 *                 + Ri(c1,c2,t)      for i = 1,2,   where
 *   R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,
 *   R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,
 *   Kv(y) = Kv0*exp(y/5) ,
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)
 * vary diurnally. The problem is posed on the square
 *   0 <= x <= 20,    30 <= y <= 50   (all in km),
 * with homogeneous Neumann boundary conditions, and for time t in
 *   0 <= t <= 86400 sec (1 day).
 * The PDE system is treated by central differences on a uniform
 * mesh, with simple polynomial initial profiles.
 *
 * The problem is solved by CVODE on NPE processors, treated
 * as a rectangular process grid of size NPEX by NPEY, with
 * NPE = NPEX*NPEY. Each processor contains a subgrid of size MXSUB
 * by MYSUB of the (x,y) mesh. Thus the actual mesh sizes are
 * MX = MXSUB*NPEX and MY = MYSUB*NPEY, and the ODE system size is
 * neq = 2*MX*MY.
 *
 * The solution is done with the BDF/GMRES method (i.e. using the
 * CVSPGMR linear solver) and a block-diagonal matrix with banded
 * blocks as a preconditioner, using the CVBBDPRE module.
 * Each block is generated using difference quotients, with
 * half-bandwidths mudq = mldq = 2*MXSUB, but the retained banded
 * blocks have half-bandwidths mukeep = mlkeep = 2.
 * A copy of the approximate Jacobian is saved and conditionally
 * reused within the preconditioner routine.
 *
 * The problem is solved twice -- with left and right preconditioning.
 *
 * Performance data and sampled solution values are printed at
 * selected output times, and all performance counters are printed
 * on completion.
 *
 * This version uses MPI for user routines.
 * Execute with number of processors = NPEX*NPEY (see constants below).
 * --------------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

```

```
#include <cvode/cvode.h>              /* prototypes for CVODE fcts. */
#include <cvode/cvode_spgmr.h>        /* prototypes and constants for CVSPGMR solver */
#include <cvode/cvode_bbdpre.h>       /* prototypes for CVBBDPRE module */
#include <nvector/nvector_parallel.h> /* definition N_Vector and macro NV_DATA_P */
#include <sundials/sundials_types.h>  /* definitions of realtype, booleantype */
#include <sundials/sundials_math.h>   /* definition of macros SQR and EXP */

#include <mpi.h>                      /* MPI constants and types */


/* Problem Constants */

#define ZERO          RCONST(0.0)

#define NVARS         2                   /* number of species          */
#define KH            RCONST(4.0e-6)      /* horizontal diffusivity Kh */
#define VEL           RCONST(0.001)       /* advection velocity V       */
#define KV0           RCONST(1.0e-8)      /* coefficient in Kv(y)       */
#define Q1            RCONST(1.63e-16)    /* coefficients q1, q2, c3    */
#define Q2            RCONST(4.66e-16)
#define C3            RCONST(3.7e16)
#define A3            RCONST(22.62)       /* coefficient in expression for q3(t) */
#define A4            RCONST(7.601)       /* coefficient in expression for q4(t) */
#define C1_SCALE      RCONST(1.0e6)       /* coefficients in initial profiles
*/
#define C2_SCALE      RCONST(1.0e12)

#define T0            ZERO                  /* initial time */
#define NOUT          12                    /* number of output times */
#define TWOHR         RCONST(7200.0)        /* number of seconds in two hours  */
#define HALFDAY       RCONST(4.32e4)        /* number of seconds in a half day */
#define PI        RCONST(3.1415926535898)  /* pi */

#define XMIN          ZERO                  /* grid boundaries in x  */
#define XMAX          RCONST(20.0)
#define YMIN          RCONST(30.0)          /* grid boundaries in y  */
#define YMAX          RCONST(50.0)

#define NPEX          2               /* no. PEs in x direction of PE array */
#define NPEY          2               /* no. PEs in y direction of PE array */
                                      /* Total no. PEs = NPEX*NPEY */
#define MXSUB         5               /* no. x points per subgrid */
#define MYSUB         5               /* no. y points per subgrid */

#define MX            (NPEX*MXSUB)    /* MX = number of x mesh points */
#define MY            (NPEY*MYSUB)    /* MY = number of y mesh points */
                                      /* Spatial mesh is MX by MY */
/* CVodeMalloc Constants */

#define RTOL    RCONST(1.0e-5)     /* scalar relative tolerance */
#define FLOOR   RCONST(100.0)      /* value of C1 or C2 at which tolerances */
                                   /* change from relative to absolute      */
#define ATOL    (RTOL*FLOOR)       /* scalar absolute tolerance */

/* Type : UserData
   contains problem constants, extended dependent variable array,
   grid constants, processor indices, MPI communicator */

typedef struct {
  realtype q4, om, dx, dy, hdco, haco, vdco;
  realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
  int my_pe, isubx, isuby;
  long int nvmxsub, nvmxsub2, Nlocal;
  MPI_Comm comm;
} *UserData;

/* Prototypes of private helper functions */

static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
                         UserData data);
static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintIntro(int npes, long int mudq, long int mldq,
                       long int mukeep, long int mlkeep);
static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        N_Vector u, realtype t);
static void PrintFinalStats(void *cvode_mem, void *pdata);
static void BSend(MPI_Comm comm,
                  int my_pe, int isubx, int isuby,
                  long int dsizex, long int dsizey,
                  realtype uarray[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[],
                      int my_pe, int isubx, int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[]);
static void BRecvWait(MPI_Request request[],
                      int isubx, int isuby,
                      long int dsizex, realtype uext[],
                      realtype buffer[]);

static void fucomm(realtype t, N_Vector u, void *f_data);

/* Prototype of function called by the solver */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);

/* Prototype of functions called by the CVBBDPRE module */

static int flocal(long int Nlocal, realtype t, N_Vector u,
                  N_Vector udot, void *f_data);

/* Private function to check function return values */

static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/***************************** Main Program ******************************/

int main(int argc, char *argv[])
{
  UserData data;
  void *cvode_mem;
  void *pdata;
  realtype abstol, reltol, t, tout;
  N_Vector u;
  int iout, my_pe, npes, flag, jpre;
  long int neq, local_N, mudq, mldq, mukeep, mlkeep;
  MPI_Comm comm;

  data = NULL;
  cvode_mem = pdata = NULL;
  u = NULL;

  /* Set problem size neq */
  neq = NVARS*MX*MY;

  /* Get processor number and total number of pe's */
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  if (npes != NPEX*NPEY) {
    if (my_pe == 0)
      fprintf(stderr, "\nMPI_ERROR(0):␣npes␣=␣%d␣is␣not␣equal␣to␣NPEX*NPEY␣=␣%d\n\n",
              npes, NPEX*NPEY);
    MPI_Finalize();
    return(1);
  }

  /* Set local length */
  local_N = NVARS*MXSUB*MYSUB;

  /* Allocate and load user data block */
  data = (UserData) malloc(sizeof *data);
  if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
  InitUserData(my_pe, local_N, comm, data);

  /* Allocate and initialize u, and set tolerances */
  u = N_VNew_Parallel(comm, local_N, neq);
  if(check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
  SetInitialProfiles(u, data);
  abstol = ATOL;
  reltol = RTOL;

  /*
     Call CVodeCreate to create the solver memory:

     CV_BDF     specifies the Backward Differentiation Formula
     CV_NEWTON  specifies a Newton iteration

     A pointer to the integrator memory is returned and stored in cvode_mem.
  */

  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
  if(check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);

  /* Set the pointer to user-defined data */
  flag = CVodeSetFdata(cvode_mem, data);
  if(check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);

  /*
     Call CVodeMalloc to initialize the integrator memory:

     cvode_mem is the pointer to the integrator memory returned by CVodeCreate
     f         is the user's right hand side function in y'=f(t,y)
     T0        is the initial time
     u         is the initial dependent variable vector
     CV_SS     specifies scalar relative and absolute tolerances
     reltol  is the relative tolerance
     &abstol is a pointer to the scalar absolute tolerance
  */

  flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
  if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);

  /* Allocate preconditioner block */
  mudq = mldq = NVARS*MXSUB;
  mukeep = mlkeep = NVARS;
  pdata = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
                         mukeep, mlkeep, ZERO, flocal, NULL);
  if(check_flag((void *)pdata, "CVBBDPrecAlloc", 0, my_pe)) MPI_Abort(comm, 1);

  /* Call CVBBDSpgmr to specify the linear solver CVSPGMR using the
     CVBBDPRE preconditioner, with left preconditioning and the
     default maximum Krylov dimension maxl  */
  flag = CVBBDSpgmr(cvode_mem, PREC_LEFT, 0, pdata);
  if(check_flag(&flag, "CVBBDSpgmr", 1, my_pe)) MPI_Abort(comm, 1);

  /* Print heading */
  if (my_pe == 0) PrintIntro(npes, mudq, mldq, mukeep, mlkeep);

  /* Loop over jpre (= PREC_LEFT, PREC_RIGHT), and solve the problem */
  for (jpre = PREC_LEFT; jpre <= PREC_RIGHT; jpre++) {

  /* On second run, re-initialize u, the integrator, CVBBDPRE, and CVSPGMR */

  if (jpre == PREC_RIGHT) {

    SetInitialProfiles(u, data);

    flag = CVodeReInit(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
    if(check_flag(&flag, "CVodeReInit", 1, my_pe)) MPI_Abort(comm, 1);

    flag = CVBBDPrecReInit(pdata, mudq, mldq, ZERO, flocal, NULL);
    if(check_flag(&flag, "CVBBDPrecReInit", 1, my_pe)) MPI_Abort(comm, 1);

    flag = CVSpilsSetPrecType(cvode_mem, PREC_RIGHT);
    check_flag(&flag, "CVSpilsSetPrecType", 1, my_pe);

    if (my_pe == 0) {
      printf("\n\n-------------------------------------------------------");
      printf("------------\n");
    }

  }


  if (my_pe == 0) {
    printf("\n\nPreconditioner␣type␣is:␣␣jpre␣=␣%s\n\n",
           (jpre == PREC_LEFT) ? "PREC_LEFT" : "PREC_RIGHT");
  }

  /* In loop over output points, call CVode, print results, test for error */

  for (iout = 1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
    if(check_flag(&flag, "CVode", 1, my_pe)) break;
    PrintOutput(cvode_mem, my_pe, comm, u, t);
  }

  /* Print final statistics */

  if (my_pe == 0) PrintFinalStats(cvode_mem, pdata);

  } /* End of jpre loop */

  /* Free memory */
  N_VDestroy_Parallel(u);
  CVBBDPrecFree(&pdata);
  free(data);
  CVodeFree(&cvode_mem);

  MPI_Finalize();

  return(0);
}

/*********************** Private Helper Functions ************************/

/* Load constants in data */

static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
                         UserData data)
{
  int isubx, isuby;

  /* Set problem constants */
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/((realtype)(MX-1));
  data->dy = (YMAX-YMIN)/((realtype)(MY-1));
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(RCONST(2.0)*data->dx);
  data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;

  /* Set machine-related constants */
  data->comm = comm;
  data->my_pe = my_pe;
  data->Nlocal = local_N;
  /* isubx and isuby are the PE grid indices corresponding to my_pe */
  isuby = my_pe/NPEX;
  isubx = my_pe - isuby*NPEX;
  data->isubx = isubx;
  data->isuby = isuby;
  /* Set the sizes of a boundary x-line in u and uext */
  data->nvmxsub = NVARS*MXSUB;
  data->nvmxsub2 = NVARS*(MXSUB+2);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, UserData data)
{
  int isubx, isuby;
  int lx, ly, jx, jy;
  long int offset;
  realtype dx, dy, x, y, cx, cy, xmid, ymid;
  realtype *uarray;

  /* Set pointer to data array in vector u */

  uarray = NV_DATA_P(u);

  /* Get mesh spacings, and subgrid indices for this PE */

  dx = data->dx;           dy = data->dy;
  isubx = data->isubx;     isuby = data->isuby;

  /* Load initial profiles of c1 and c2 into local u vector.
  Here lx and ly are local mesh point indices on the local subgrid,
  and jx and jy are the global mesh point indices. */

  offset = 0;
  xmid = RCONST(0.5)*(XMIN + XMAX);
  ymid = RCONST(0.5)*(YMIN + YMAX);
  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    y = YMIN + jy*dy;
    cy = SQR(RCONST(0.1)*(y - ymid));
    cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      x = XMIN + jx*dx;
      cx = SQR(RCONST(0.1)*(x - xmid));
      cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
      uarray[offset  ] = C1_SCALE*cx*cy;
      uarray[offset+1] = C2_SCALE*cx*cy;
      offset = offset + 2;
    }
  }
}

/* Print problem introduction */

static void PrintIntro(int npes, long int mudq, long int mldq,
                       long int mukeep, long int mlkeep)
{
  printf("\n2-species␣diurnal␣advection-diffusion␣problem\n");
  printf("␣␣%d␣by␣%d␣mesh␣on␣%d␣processors\n", MX, MY, npes);
  printf("␣␣Using␣CVBBDPRE␣preconditioner␣module\n");
  printf("␣␣␣␣Difference-quotient␣half-bandwidths␣are");
  printf("␣mudq␣=␣%ld,␣␣mldq␣=␣%ld\n", mudq, mldq);
  printf("␣␣␣␣Retained␣band␣block␣half-bandwidths␣are");
  printf("␣mukeep␣=␣%ld,␣␣mlkeep␣=␣%ld", mukeep, mlkeep);

  return;
}

/* Print current t, step count, order, stepsize, and sampled c1,c2 values */

static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
                        N_Vector u, realtype t)
{
  int qu, flag, npelast;
  long int i0, i1, nst;
  realtype hu, *uarray, tempu[2];
  MPI_Status status;

  npelast = NPEX*NPEY - 1;
  uarray = NV_DATA_P(u);

  /* Send c1,c2 at top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NVARS*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
      MPI_Send(&uarray[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      tempu[0] = uarray[i0];
      tempu[1] = uarray[i1];
    }
  }

  /* On PE 0, receive c1,c2 at top right, then print performance data
     and sampled solution values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    flag = CVodeGetNumSteps(cvode_mem, &nst);
    check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
    flag = CVodeGetLastOrder(cvode_mem, &qu);
    check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
    flag = CVodeGetLastStep(cvode_mem, &hu);
    check_flag(&flag, "CVodeGetLastStep", 1, my_pe);
#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("t␣=␣%.2Le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2Le\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3Le␣%12.3Le␣\n", uarray[0], uarray[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3Le␣%12.3Le␣\n\n", tempu[0], tempu[1]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("t␣=␣%.2le␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2le\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3le␣%12.3le␣\n", uarray[0], uarray[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3le␣%12.3le␣\n\n", tempu[0], tempu[1]);
#else
    printf("t␣=␣%.2e␣␣␣no.␣steps␣=␣%ld␣␣␣order␣=␣%d␣␣␣stepsize␣=␣%.2e\n",
           t, nst, qu, hu);
    printf("At␣bottom␣left:␣␣c1,␣c2␣=␣%12.3e␣%12.3e␣\n", uarray[0], uarray[1]);
    printf("At␣top␣right:␣␣␣␣c1,␣c2␣=␣%12.3e␣%12.3e␣\n\n", tempu[0], tempu[1]);
#endif
  }
}

/* Print final statistics contained in iopt */

static void PrintFinalStats(void *cvode_mem, void *pdata)
{
  long int lenrw, leniw ;
  long int lenrwLS, leniwLS;
  long int lenrwBBDP, leniwBBDP, ngevalsBBDP;
  long int nst, nfe, nsetups, nni, ncfn, netf;
  long int nli, npe, nps, ncfl, nfeLS;
  int flag;

  flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
  check_flag(&flag, "CVodeGetWorkSpace", 1, 0);
  flag = CVodeGetNumSteps(cvode_mem, &nst);
  check_flag(&flag, "CVodeGetNumSteps", 1, 0);
  flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
  check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
  flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
  check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
  flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
  check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
  flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
  check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
  flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
  check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);

  flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
  check_flag(&flag, "CVSpilsGetWorkSpace", 1, 0);
  flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
  check_flag(&flag, "CVSpilsGetNumLinIters", 1, 0);
  flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
  check_flag(&flag, "CVSpilsGetNumPrecEvals", 1, 0);
  flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
  check_flag(&flag, "CVSpilsGetNumPrecSolves", 1, 0);
  flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
  check_flag(&flag, "CVSpilsGetNumConvFails", 1, 0);
  flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
  check_flag(&flag, "CVSpilsGetNumRhsEvals", 1, 0);

  printf("\nFinal␣Statistics:␣\n\n");
  printf("lenrw␣␣␣=␣%5ld␣␣␣␣␣leniw␣␣␣=␣%5ld\n", lenrw, leniw);
  printf("lenrwls␣=␣%5ld␣␣␣␣␣leniwls␣=␣%5ld\n", lenrwLS, leniwLS);
  printf("nst␣␣␣␣␣=␣%5ld\n"                  , nst);
  printf("nfe␣␣␣␣␣=␣%5ld␣␣␣␣␣nfels␣␣␣=␣%5ld\n"  , nfe, nfeLS);
  printf("nni␣␣␣␣␣=␣%5ld␣␣␣␣␣nli␣␣␣␣␣=␣%5ld\n"  , nni, nli);
  printf("nsetups␣=␣%5ld␣␣␣␣␣netf␣␣␣␣=␣%5ld\n"  , nsetups, netf);
  printf("npe␣␣␣␣␣=␣%5ld␣␣␣␣␣nps␣␣␣␣␣=␣%5ld\n"  , npe, nps);
  printf("ncfn␣␣␣␣=␣%5ld␣␣␣␣␣ncfl␣␣␣␣=␣%5ld\n\n", ncfn, ncfl);

  flag = CVBBDPrecGetWorkSpace(pdata, &lenrwBBDP, &leniwBBDP);
  check_flag(&flag, "CVBBDPrecGetWorkSpace", 1, 0);
  flag = CVBBDPrecGetNumGfnEvals(pdata, &ngevalsBBDP);
  check_flag(&flag, "CVBBDPrecGetNumGfnEvals", 1, 0);
  printf("In␣CVBBDPRE:␣real/integer␣local␣work␣space␣sizes␣=␣%ld,␣%ld\n",
         lenrwBBDP, leniwBBDP);
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣no.␣flocal␣evals.␣=␣%ld\n",ngevalsBBDP);
}

/* Routine to send boundary data to neighboring PEs */

static void BSend(MPI_Comm comm,
                  int my_pe, int isubx, int isuby,
                  long int dsizex, long int dsizey,
                  realtype uarray[])
{
  int i, ly;
  long int offsetu, offsetbuf;
  realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];

  /* If isuby > 0, send data from bottom x-line of u */

  if (isuby != 0)
    MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If isuby < NPEY-1, send data from top x-line of u */

  if (isuby != NPEY-1) {
    offsetu = (MYSUB-1)*dsizex;
    MPI_Send(&uarray[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
  }

  /* If isubx > 0, send data from left y-line of u (via bufleft) */

  if (isubx != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = ly*dsizex;
      for (i = 0; i < NVARS; i++)
        bufleft[offsetbuf+i] = uarray[offsetu+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */

  if (isubx != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
      for (i = 0; i < NVARS; i++)
        bufright[offsetbuf+i] = uarray[offsetu+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }

}

/* Routine to start receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvPost(MPI_Comm comm, MPI_Request request[],
                      int my_pe, int isubx, int isuby,
                      long int dsizex, long int dsizey,
                      realtype uext[], realtype buffer[])
{
  long int offsetue;
  /* Have bufleft and bufright use the same buffer */
  realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
                                              my_pe-NPEX, 0, comm, &request[0]);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1) {
    offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
                                         my_pe+NPEX, 0, comm, &request[1]);
  }

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                         my_pe-1, 0, comm, &request[2]);
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                         my_pe+1, 0, comm, &request[3]);
  }

}

/* Routine to finish receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvWait(MPI_Request request[],
                      int isubx, int isuby,
                      long int dsizex, realtype uext[],
                      realtype buffer[])
{
  int i, ly;
  long int dsizex2, offsetue, offsetbuf;
  realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NVARS;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Wait(&request[0],&status);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+1)*dsizex2;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufleft[offsetbuf+i];
    }
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Wait(&request[3],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+2)*dsizex2 - NVARS;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufright[offsetbuf+i];
    }
  }
}

/* fucomm routine.  This routine performs all inter-processor
   communication of data in u needed to calculate f.          */

static void fucomm(realtype t, N_Vector u, void *f_data)
{
  UserData data;
  realtype *uarray, *uext, buffer[2*NVARS*MYSUB];
  MPI_Comm comm;
  int my_pe, isubx, isuby;
  long int nvmxsub, nvmysub;
  MPI_Request request[4];

  data = (UserData) f_data;
  uarray = NV_DATA_P(u);

  /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */

  comm = data->comm;  my_pe = data->my_pe;
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;
  nvmysub = NVARS*MYSUB;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs */

  BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs */

  BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, uarray);

  /* Finish receiving boundary data from neighboring PEs */

  BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
}

/***************** Function called by the solver **************************/

/* f routine.  Evaluate f(t,y).  First call fucomm to do communication of
   subgrid boundary data into uext.  Then calculate f by a call to flocal. */

static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
{
  UserData data;

  data = (UserData) f_data;

  /* Call fucomm to do inter-processor communication */

  fucomm (t, u, f_data);

  /* Call flocal to calculate all right-hand sides */

  flocal (data->Nlocal, t, u, udot, f_data);

  return(0);
}

/***************** Functions called by the CVBBDPRE module ****************/

/* flocal routine.  Compute f(t,y).  This routine assumes that all
   inter-processor communication of data needed to calculate f has already
   been done, and this data is in the work array uext.                    */

static int flocal(long int Nlocal, realtype t, N_Vector u,
                  N_Vector udot, void *f_data)
{
  realtype *uext;
  realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
  realtype q4coef, dely, verdco, hordco, horaco;
  int i, lx, ly, jx, jy;
  int isubx, isuby;
  long int nvmxsub, nvmxsub2, offsetu, offsetue;
  UserData data;
  realtype *uarray, *duarray;

  uarray = NV_DATA_P(u);
  duarray = NV_DATA_P(udot);

  /* Get subgrid indices, array sizes, extended work array uext */

  data = (UserData) f_data;
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
  uext = data->uext;

  /* Copy local segment of u vector into the working extended array uext */

  offsetu = 0;
  offsetue = nvmxsub2 + NVARS;
  for (ly = 0; ly < MYSUB; ly++) {
    for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = uarray[offsetu+i];
    offsetu = offsetu + nvmxsub;
    offsetue = offsetue + nvmxsub2;
  }

  /* To facilitate homogeneous Neumann boundary conditions, when this is
  a boundary PE, copy data from the first interior mesh line of u to uext */

  /* If isuby = 0, copy x-line 2 of u to uext */
  if (isuby == 0) {
    for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = uarray[nvmxsub+i];
  }

  /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
  if (isuby == NPEY-1) {
    offsetu = (MYSUB-2)*nvmxsub;
    offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
    for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = uarray[offsetu+i];
  }

  /* If isubx = 0, copy y-line 2 of u to uext */
  if (isubx == 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*nvmxsub + NVARS;
      offsetue = (ly+1)*nvmxsub2;
      for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
    }
  }

  /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
  if (isubx == NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = (ly+1)*nvmxsub - 2*NVARS;
      offsetue = (ly+2)*nvmxsub2 - NVARS;
      for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
    }
  }

  /* Make local copies of problem variables, for efficiency */

  dely = data->dy;
  verdco = data->vdco;
  hordco = data->hdco;
  horaco = data->haco;

  /* Set diurnal rate coefficients as functions of t, and save q4 in
  data block for use by preconditioner evaluation routine            */

  s = sin((data->om)*t);
  if (s > ZERO) {
    q3 = EXP(-A3/s);
    q4coef = EXP(-A4/s);
  } else {
    q3 = ZERO;
    q4coef = ZERO;
  }
  data->q4 = q4coef;


  /* Loop over all grid points in local subgrid */

  for (ly = 0; ly < MYSUB; ly++) {

    jy = ly + isuby*MYSUB;

    /* Set vertical diffusion coefficients at jy +- 1/2 */

    ydn = YMIN + (jy - RCONST(0.5))*dely;
    yup = ydn + dely;
    cydn = verdco*EXP(RCONST(0.2)*ydn);
    cyup = verdco*EXP(RCONST(0.2)*yup);
    for (lx = 0; lx < MXSUB; lx++) {

      jx = lx + isubx*MXSUB;

      /* Extract c1 and c2, and set kinetic rate terms */

      offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
      c1 = uext[offsetue];
      c2 = uext[offsetue+1];
      qq1 = Q1*c1*C3;
      qq2 = Q2*c1*c2;
      qq3 = q3*C3;
      qq4 = q4coef*c2;
      rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
      rkin2 = qq1 - qq2 - qq4;

      /* Set vertical diffusion terms */

      c1dn = uext[offsetue-nvmxsub2];
      c2dn = uext[offsetue-nvmxsub2+1];
      c1up = uext[offsetue+nvmxsub2];
      c2up = uext[offsetue+nvmxsub2+1];
      vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
      vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);

      /* Set horizontal diffusion and advection terms */

      c1lt = uext[offsetue-2];
      c2lt = uext[offsetue-1];
      c1rt = uext[offsetue+2];
      c2rt = uext[offsetue+3];
      hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
      hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
      horad1 = horaco*(c1rt - c1lt);
      horad2 = horaco*(c2rt - c2lt);

      /* Load all terms into duarray */

      offsetu = lx*NVARS + ly*nvmxsub;
      duarray[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
      duarray[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
    }
  }

  return(0);
}

/* Check function return value...
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns a flag so check if
              flag >= 0
     opt == 2 means function allocates memory so check if returned
              NULL pointer */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr, "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
            id, funcname);
    return(1); }

  /* Check if flag < 0 */
  else if (opt == 1) {
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣with␣flag␣=␣%d\n\n",
              id, funcname, *errflag);
      return(1); }}

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && flagvalue == NULL) {
    fprintf(stderr, "\nMEMORY_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
            id, funcname);
    return(1); }

  return(0);
}
```

G Listing of fcvkryx.f

```
C     ----------------------------------------------------------------
C     $Revision: 1.1 $
C     $Date: 2006/07/05 15:50:04 $
C     ----------------------------------------------------------------
C     FCVODE Example Problem: 2D kinetics-transport, precond. Krylov
C     solver.
C
C     An ODE system is generated from the following 2-species diurnal
C     kinetics advection-diffusion PDE system in 2 space dimensions:
C
C     dc(i)/dt = Kh*(d/dx)**2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)
C                          + Ri(c1,c2,t)      for i = 1,2,   where
C     R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,
C     R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,
C     Kv(y) = Kv0*exp(y/5) ,
C     Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)
C     vary diurnally.
C
C     The problem is posed on the square
C     0 .le. x .le. 20,    30 .le. y .le. 50   (all in km),
C     with homogeneous Neumann boundary conditions, and for time t
C     in 0 .le. t .le. 86400 sec (1 day).
C     The PDE system is treated by central differences on a uniform
C     10 x 10 mesh, with simple polynomial initial profiles.
C     The problem is solved with CVODE, with the BDF/GMRES method and
C     the block-diagonal part of the Jacobian as a left
C     preconditioner.
C
C     Note: this program requires the dense linear solver routines
C     DGEFA and DGESL from LINPACK, and BLAS routines DCOPY and DSCAL.
C
C     The second and third dimensions of U here must match the values
C     of MX and MY, for consistency with the output statements
C     below.
C     ----------------------------------------------------------------
C
      IMPLICIT NONE
C
      INTEGER*4 MX, MY, NEQ
      PARAMETER (MX=10, MY=10)
      PARAMETER (NEQ=2*MX*MY)
      INTEGER*4 LENIPAR, LENRPAR
      PARAMETER (LENIPAR=6+2*MX*MY, LENRPAR=12+8*MX*MY)
C
      INTEGER METH,ITMETH,IATOL,ITASK,IER,LNCFL,LNPS
      INTEGER LNST,LNFE,LNSETUP,LNNI,LNCF,LQ,LH,LNPE,LNLI,LNETF
      INTEGER JOUT,JPRETYPE,IGSTYPE,MAXL
      INTEGER*4 IOUT(25),IPAR(LENIPAR)
      INTEGER*4 NST,NFE,NPSET,NPE,NPS,NNI,NETF
      INTEGER*4 NLI,NCFN,NCFL
      DOUBLE PRECISION ATOL,AVDIM,T,TOUT,TWOHR,RTOL,FLOOR,DELT
      DOUBLE PRECISION U(2,MX,MY),ROUT(10),RPAR(LENRPAR)
C
      DATA TWOHR/7200.0D0/, RTOL/1.0D-5/, FLOOR/100.0D0/,
     &     JPRETYPE/1/, IGSTYPE/1/, MAXL/0/, DELT/0.0D0/
      DATA LNST/3/, LNFE/4/, LNETF/5/,  LNCF/6/, LNNI/7/, LNSETUP/8/,
     &     LQ/9/, LNPE/18/, LNLI/20/, LNPS/19/, LNCFL/21/
```

```
      DATA LH/2/
C
C     Load problem constants into IPAR, RPAR, and set initial values
      CALL INITKX(MX, MY, U, IPAR, RPAR)
C
C     Set other input arguments.
      T = 0.0D0
      METH = 2
      ITMETH = 2
      IATOL = 1
      ATOL = RTOL * FLOOR
      ITASK = 1
C
      WRITE(6,10) NEQ
 10   FORMAT('Krylov example problem:'//
     &       ' Kinetics-transport, NEQ = ', I4/)
C
      CALL FNVINITS(1, NEQ, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,20) IER
 20     FORMAT(///' SUNDIALS_ERROR: FNVINITS returned IER = ', I5)
        STOP
      ENDIF
C
C     Initialize CVODE
      CALL FCVMALLOC(T, U, METH, ITMETH, IATOL, RTOL, ATOL,
     &     IOUT, ROUT, IPAR, RPAR, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,30) IER
 30     FORMAT(///' SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
        STOP
        ENDIF
C
      CALL FCVSPGMR(JPRETYPE, IGSTYPE, MAXL, DELT, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,40) IER
 40     FORMAT(///' SUNDIALS_ERROR: FCVSPGMR returned IER = ', I5)
        CALL FCVFREE
        STOP
      ENDIF
C
      CALL FCVSPILSSETPREC(1, IER)
C
C Loop over output points, call FCVODE, print sample solution values.
      TOUT = TWOHR
      DO JOUT = 1, 12
C
         CALL FCVODE(TOUT, T, U, ITASK, IER)
C
         WRITE(6,50) T, IOUT(LNST), IOUT(LQ), ROUT(LH)
 50      FORMAT(/' t = ', E11.3, 3X, 'nst = ', I5,
     &           '  q = ', I2, '  h = ', E14.6)
         WRITE(6,55) U(1,1,1), U(1,5,5), U(1,10,10),
     &               U(2,1,1), U(2,5,5), U(2,10,10)
 55      FORMAT('  c1 (bot.left/middle/top rt.) = ', 3E14.6/
     &          '  c2 (bot.left/middle/top rt.) = ', 3E14.6)
C
         IF (IER .NE. 0) THEN
            WRITE(6,60) IER, IOUT(15)
 60         FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
     &             '                 Linear Solver returned IER = ', I5)
            CALL FCVFREE
            STOP
         ENDIF
C
         TOUT = TOUT + TWOHR
C
      ENDDO

C     Print final statistics.
      NST = IOUT(LNST)
      NFE = IOUT(LNFE)
      NPSET = IOUT(LNSETUP)
      NPE = IOUT(LNPE)
      NPS = IOUT(LNPS)
      NNI = IOUT(LNNI)
      NLI = IOUT(LNLI)
      AVDIM = DBLE(NLI) / DBLE(NNI)
      NCFN = IOUT(LNCF)
      NCFL = IOUT(LNCFL)
      NETF = IOUT(LNETF)
      WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
     &     NCFL, NETF
  80  FORMAT(//'Final statistics:'//
     &     ' number of steps        = ', I5, 5X,
     &     ' number of f evals.     =', I5/
     &     ' number of prec. setups = ', I5/
     &     ' number of prec. evals. = ', I5, 5X,
     &     ' number of prec. solves = ', I5/
     &     ' number of nonl. iters. = ', I5, 5X,
     &     ' number of lin. iters.  = ', I5/
     &     ' average Krylov subspace dimension (NLI/NNI)  = ', E14.6/
     &     ' number of conv. failures.. nonlinear = ', I3,
     &     ' linear = ', I3/
     &     ' number of error test failures = ', I3)
C
      CALL FCVFREE
C
      STOP
      END

C     ----------------------------------------------------------------

      SUBROUTINE INITKX(MX, MY, U0, IPAR, RPAR)
C     Routine to set problem constants and initial values
C
      IMPLICIT NONE
C
      INTEGER*4 MX, MY, IPAR(*)
      DOUBLE PRECISION RPAR(*)
C
      INTEGER*4 MM, JY, JX, P_IPP, P_BD, P_P
      DOUBLE PRECISION U0
      DIMENSION U0(2,MX,MY)
      DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
      DOUBLE PRECISION VDCO, HACO, X, Y
      DOUBLE PRECISION CX, CY, DKH, DKV0, DX, HALFDA, PI, VEL
C
```

```
      DATA DKH/4.0D-6/, VEL/0.001D0/, DKV0/1.0D-8/, HALFDA/4.32D4/,
     1     PI/3.1415926535898D0/
C
C      Problem constants
      MM = MX * MY
      Q1 = 1.63D-16
      Q2 = 4.66D-16
      A3 = 22.62D0
      A4 = 7.601D0
      OM = PI / HALFDA
      C3 = 3.7D16
      DX = 20.0D0 / (MX - 1.0D0)
      DY = 20.0D0 / (MY - 1.0D0)
      HDCO = DKH / DX**2
      HACO = VEL / (2.0D0 * DX)
      VDCO = (1.0D0 / DY**2) * DKV0
C
C      Load constants in IPAR and RPAR
      IPAR(1) = MX
      IPAR(2) = MY
      IPAR(3) = MM
C
      RPAR(1)  = Q1
      RPAR(2)  = Q2
      RPAR(3)  = Q3
      RPAR(4)  = Q4
      RPAR(5)  = A3
      RPAR(6)  = A4
      RPAR(7)  = OM
      RPAR(8)  = C3
      RPAR(9)  = DY
      RPAR(10) = HDCO
      RPAR(11) = VDCO
      RPAR(12) = HACO
C
C      Pointers into IPAR and RPAR
      P_IPP  = 7
      P_BD   = 13
      P_P    = P_BD + 4*MM
C
      IPAR(4) = P_IPP
      IPAR(5) = P_BD
      IPAR(6) = P_P
C
C      Set initial profiles.
      DO JY = 1, MY
         Y = 30.0D0 + (JY - 1.0D0) * DY
         CY = (0.1D0 * (Y - 40.0D0))**2
         CY = 1.0D0 - CY + 0.5D0 * CY**2
         DO JX = 1, MX
            X = (JX - 1.0D0) * DX
            CX = (0.1D0 * (X - 10.0D0))**2
            CX = 1.0D0 - CX + 0.5D0 * CX**2
            U0(1,JX,JY) = 1.0D6 * CX * CY
            U0(2,JX,JY) = 1.0D12 * CX * CY
         ENDDO
      ENDDO
C
      RETURN
      END

C     ----------------------------------------------------------------

      SUBROUTINE FCVFUN(T, U, UDOT, IPAR, RPAR, IER)
C     Routine for right-hand side function f
C
      IMPLICIT NONE
C
      DOUBLE PRECISION T, U(2,*), UDOT(2,*), RPAR(*)
      INTEGER*4 IPAR(*), IER
C
      INTEGER ILEFT, IRIGHT
      INTEGER*4 JX, JY, MX, MY, MM, IBLOK0, IBLOK, IDN, IUP
      DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
      DOUBLE PRECISION VDCO, HACO
      DOUBLE PRECISION C1, C2, C1DN, C2DN, C1UP, C2UP, C1LT, C2LT
      DOUBLE PRECISION C1RT, C2RT, CYDN, CYUP, HORD1, HORD2, HORAD1
      DOUBLE PRECISION HORAD2, QQ1, QQ2, QQ3, QQ4, RKIN1, RKIN2, S
      DOUBLE PRECISION VERTD1, VERTD2, YDN, YUP
C
C     Extract constants from IPAR and RPAR
      MX = IPAR(1)
      MY = IPAR(2)
      MM = IPAR(3)
C
      Q1 = RPAR(1)
      Q2 = RPAR(2)
      Q3 = RPAR(3)
      Q4 = RPAR(4)
      A3 = RPAR(5)
      A4 = RPAR(6)
      OM = RPAR(7)
      C3 = RPAR(8)
      DY = RPAR(9)
      HDCO = RPAR(10)
      VDCO = RPAR(11)
      HACO = RPAR(12)
C
C     Set diurnal rate coefficients.
      S = SIN(OM * T)
      IF (S .GT. 0.0D0) THEN
         Q3 = EXP(-A3 / S)
         Q4 = EXP(-A4 / S)
      ELSE
         Q3 = 0.0D0
         Q4 = 0.0D0
      ENDIF
      RPAR(3) = Q3
      RPAR(4) = Q4
C
C     Loop over all grid points.
      DO JY = 1, MY
         YDN = 30.0D0 + (JY - 1.5D0) * DY
         YUP = YDN + DY
         CYDN = VDCO * EXP(0.2D0 * YDN)
         CYUP = VDCO * EXP(0.2D0 * YUP)
         IBLOK0 = (JY - 1) * MX
         IDN = -MX
```

```
            IF (JY .EQ. 1) IDN = MX
            IUP = MX
            IF (JY .EQ. MY) IUP = -MX
            DO JX = 1, MX
               IBLOK = IBLOK0 + JX
               C1 = U(1,IBLOK)
               C2 = U(2,IBLOK)
C      Set kinetic rate terms.
               QQ1 = Q1 * C1 * C3
               QQ2 = Q2 * C1 * C2
               QQ3 = Q3 * C3
               QQ4 = Q4 * C2
               RKIN1 = -QQ1 - QQ2 + 2.0D0 * QQ3 + QQ4
               RKIN2 = QQ1 - QQ2 - QQ4
C      Set vertical diffusion terms.
               C1DN = U(1,IBLOK + IDN)
               C2DN = U(2,IBLOK + IDN)
               C1UP = U(1,IBLOK + IUP)
               C2UP = U(2,IBLOK + IUP)
               VERTD1 = CYUP * (C1UP - C1) - CYDN * (C1 - C1DN)
               VERTD2 = CYUP * (C2UP - C2) - CYDN * (C2 - C2DN)
C      Set horizontal diffusion and advection terms.
               ILEFT = -1
               IF (JX .EQ. 1) ILEFT = 1
               IRIGHT = 1
               IF (JX .EQ. MX) IRIGHT = -1
               C1LT = U(1,IBLOK + ILEFT)
               C2LT = U(2,IBLOK + ILEFT)
               C1RT = U(1,IBLOK + IRIGHT)
               C2RT = U(2,IBLOK + IRIGHT)
               HORD1 = HDCO * (C1RT - 2.0D0 * C1 + C1LT)
               HORD2 = HDCO * (C2RT - 2.0D0 * C2 + C2LT)
               HORAD1 = HACO * (C1RT - C1LT)
               HORAD2 = HACO * (C2RT - C2LT)
C      Load all terms into UDOT.
               UDOT(1,IBLOK) = VERTD1 + HORD1 + HORAD1 + RKIN1
               UDOT(2,IBLOK) = VERTD2 + HORD2 + HORAD2 + RKIN2
            ENDDO
         ENDDO
C
         IER = 0
C
         RETURN
         END

C      ----------------------------------------------------------------

         SUBROUTINE FCVPSET(T, U, FU, JOK, JCUR, GAMMA, H,
        &                   IPAR, RPAR, V1, V2, V3, IER)
C      Routine to set and preprocess block-diagonal preconditioner.
C      Note: The dimensions in /BDJ/ below assume at most 100 mesh points.
C
         IMPLICIT NONE
C
         INTEGER IER, JOK, JCUR
         DOUBLE PRECISION T, U(2,*), FU(*), GAMMA, H
         INTEGER*4 IPAR(*)
         DOUBLE PRECISION RPAR(*), V1(*), V2(*), V3(*)
C
```

```
      INTEGER*4 MX, MY, MM, P_IPP, P_BD, P_P
      DOUBLE PRECISION Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO
C
      IER = 0
C
C     Extract constants from IPAR and RPAR
      MX = IPAR(1)
      MY = IPAR(2)
      MM = IPAR(3)
C
      Q1 = RPAR(1)
      Q2 = RPAR(2)
      Q3 = RPAR(3)
      Q4 = RPAR(4)
      C3 = RPAR(8)
      DY = RPAR(9)
      HDCO = RPAR(10)
      VDCO = RPAR(11)
C
C     Extract pointers into IPAR and RPAR
      P_IPP = IPAR(4)
      P_BD  = IPAR(5)
      P_P   = IPAR(6)
C
C     If needed, recompute BD
C
      IF (JOK .EQ. 1) THEN
C     JOK = 1. Use saved BD
        JCUR = 0
      ELSE
C     JOK = 0. Compute diagonal Jacobian blocks.
C     (using q4 value computed on last FCVFUN call).
         CALL PREC_JAC(MX, MY, MM, U, RPAR(P_BD),
     &         Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO)
         JCUR = 1
      ENDIF
C
C     Copy BD to P
      CALL DCOPY(4*MM, RPAR(P_BD), 1, RPAR(P_P), 1)
C
C     Scale P by -GAMMA
      CALL DSCAL(4*MM, -GAMMA, RPAR(P_P), 1)
C
C     Perform LU decomposition
      CALL PREC_LU(MM, RPAR(P_P), IPAR(P_IPP), IER)
C
      RETURN
      END

C     ----------------------------------------------------------------

      SUBROUTINE FCVPSOL(T, U, FU, R, Z, GAMMA, DELTA, LR,
     &                   IPAR, RPAR, VTEMP, IER)
C     Routine to solve preconditioner linear system.
C
      IMPLICIT NONE
C
      INTEGER IER, LR
      INTEGER*4 IPAR(*)
      DOUBLE PRECISION T, U(*), FU(*), R(*), Z(2,*)
      DOUBLE PRECISION GAMMA, DELTA, RPAR(*)
      DOUBLE PRECISION VTEMP(*)
C
      INTEGER*4 MM, P_IPP, P_P
C
      IER = 0
C
C     Extract constants from IPAR and RPAR
      MM   = IPAR(3)
C
C     Extract pointers into IPAR and RPAR
      P_IPP = IPAR(4)
      P_P   = IPAR(6)
C
C     Copy RHS into Z
      CALL DCOPY(2*MM, R, 1, Z, 1)
C
C     Solve the block-diagonal system Px = r using LU factors stored in P
C     and pivot data in IPP, and return the solution in Z.
      CALL PREC_SOL(MM, RPAR(P_P), IPAR(P_IPP), Z)

      RETURN
      END

C     -----------------------------------------------------------------

      SUBROUTINE PREC_JAC(MX, MY, MM, U, BD,
     &     Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO)
C     Routine to compute diagonal Jacobian blocks
C
      IMPLICIT NONE
C
      INTEGER*4 MX, MY, MM
      DOUBLE PRECISION U(2,*), BD(2,2,MM)
      DOUBLE PRECISION Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO
C
      INTEGER*4 JY, JX, IBLOK, IBLOK0
      DOUBLE PRECISION C1, C2, CYDN, CYUP, DIAG, YDN, YUP
C
      DO JY = 1, MY
         YDN = 30.0D0 + (JY - 1.5D0) * DY
         YUP = YDN + DY
         CYDN = VDCO * EXP(0.2D0 * YDN)
         CYUP = VDCO * EXP(0.2D0 * YUP)
         DIAG = -(CYDN + CYUP + 2.0D0 * HDCO)
         IBLOK0 = (JY - 1) * MX
         DO JX = 1, MX
            IBLOK = IBLOK0 + JX
            C1 = U(1,IBLOK)
            C2 = U(2,IBLOK)
            BD(1,1,IBLOK) = (-Q1 * C3 - Q2 * C2) + DIAG
            BD(1,2,IBLOK) = -Q2 * C1 + Q4
            BD(2,1,IBLOK) =  Q1 * C3 - Q2 * C2
            BD(2,2,IBLOK) = (-Q2 * C1 - Q4) + DIAG
         ENDDO
      ENDDO

      RETURN
      END

C     ----------------------------------------------------------------

      SUBROUTINE PREC_LU(MM, P, IPP, IER)
C     Routine to perform LU decomposition on (P+I)
C
      IMPLICIT NONE
C
      INTEGER IER
      INTEGER*4 MM, IPP(2,MM)
      DOUBLE PRECISION P(2,2,MM)
C
      INTEGER*4 I
C
C     Add identity matrix and do LU decompositions on blocks, in place.
      DO I = 1, MM
         P(1,1,I) = P(1,1,I) + 1.0D0
         P(2,2,I) = P(2,2,I) + 1.0D0
         CALL DGEFA(P(1,1,I), 2, 2, IPP(1,I), IER)
         IF (IER .NE. 0) RETURN
      ENDDO
C
      RETURN
      END

C     ----------------------------------------------------------------

      SUBROUTINE PREC_SOL(MM, P, IPP, Z)
C     Routine for backsolve
C
      IMPLICIT NONE
C
      INTEGER*4 MM, IPP(2,MM)
      DOUBLE PRECISION P(2,2,MM), Z(2,MM)
C
      INTEGER*4 I
C
      DO I = 1, MM
         CALL DGESL(P(1,1,I), 2, 2, IPP(1,I), Z(1,I), 0)
      ENDDO

      RETURN
      END

C     ----------------------------------------------------------------

      subroutine dgefa(a, lda, n, ipvt, info)
c
      implicit none
c
      integer info, idamax, j, k, kp1, l, nm1, n
      integer*4 lda, ipvt(1)
      double precision a(lda,1), t
c
c     dgefa factors a double precision matrix by gaussian elimination.
c
c     dgefa is usually called by dgeco, but it can be called
c     directly with a saving in time if  rcond  is not needed.
```

```
c     (time for dgeco) = (1 + 9/n)*(time for dgefa) .
c
c     on entry
c
c        a       double precision(lda, n)
c                the matrix to be factored.
c
c        lda     integer
c                the leading dimension of the array  a .
c
c        n       integer
c                the order of the matrix  a .
c
c     on return
c
c        a       an upper triangular matrix and the multipliers
c                which were used to obtain it.
c                the factorization can be written  a = l*u  where
c                l  is a product of permutation and unit lower
c                triangular matrices and  u  is upper triangular.
c
c        ipvt    integer(n)
c                an integer vector of pivot indices.
c
c        info    integer
c                = 0  normal value.
c                = k  if  u(k,k) .eq. 0.0 .  this is not an error
c                     condition for this subroutine, but it does
c                     indicate that dgesl or dgedi will divide by zero
c                     if called.  use  rcond  in dgeco for a reliable
c                     indication of singularity.
c
c     linpack. this version dated 08/14/78 .
c     cleve moler, university of new mexico, argonne national lab.
c
c     subroutines and functions
c
c     blas daxpy,dscal,idamax
c
c     internal variables
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamax(n - k + 1, a(k,k), 1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
c
c           interchange if necessary
```

```
c
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
c
c           compute multipliers
c
            t = -1.0d0 / a(k,k)
            call dscal(n - k, t, a(k + 1,k), 1)
c
c           row elimination with column indexing
c
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call daxpy(n - k, t, a(k + 1,k), 1, a(k + 1,j), 1)
   30       continue
         go to 50
   40    continue
            info = k
   50    continue
   60 continue
   70 continue
      ipvt(n) = n
      if (a(n,n) .eq. 0.0d0) info = n
      return
      end

C     ----------------------------------------------------------------

      subroutine dgesl(a, lda, n, ipvt, b, job)
c
      implicit none
c
      integer lda, n, job, k, kb, l, nm1
      integer*4 ipvt(1)
      double precision a(lda,1), b(1), ddot, t
c
c     dgesl solves the double precision system
c     a * x = b  or  trans(a) * x = b
c     using the factors computed by dgeco or dgefa.
c
c     on entry
c
c        a       double precision(lda, n)
c                the output from dgeco or dgefa.
c
c        lda     integer
c                the leading dimension of the array  a .
c
c        n       integer
c                the order of the matrix  a .
c
c        ipvt    integer(n)
```

```
c                 the pivot vector from dgeco or dgefa.
c
c        b        double precision(n)
c                 the right hand side vector.
c
c        job      integer
c                 = 0         to solve  a*x = b ,
c                 = nonzero   to solve  trans(a)*x = b  where
c                             trans(a)  is the transpose.
c
c     on return
c
c        b        the solution vector  x .
c
c     error condition
c
c        a division by zero will occur if the input factor contains a
c        zero on the diagonal.  technically this indicates singularity
c        but it is often caused by improper arguments or improper
c        setting of lda .  it will not occur if the subroutines are
c        called correctly and if dgeco has set rcond .gt. 0.0
c        or dgefa has set info .eq. 0 .
c
c     to compute  inverse(a) * c  where  c  is a matrix
c     with  p  columns
c           call dgeco(a,lda,n,ipvt,rcond,z)
c           if (rcond is too small) go to ...
c           do 10 j = 1, p
c              call dgesl(a,lda,n,ipvt,c(1,j),0)
c        10 continue
c
c     linpack. this version dated 08/14/78 .
c     cleve moler, university of new mexico, argonne national lab.
c
c     subroutines and functions
c
c     blas daxpy,ddot
c
c     internal variables
c
      nm1 = n - 1
      if (job .ne. 0) go to 50
c
c        job = 0 , solve  a * x = b
c        first solve  l*y = b
c
         if (nm1 .lt. 1) go to 30
         do 20 k = 1, nm1
            l = ipvt(k)
            t = b(l)
            if (l .eq. k) go to 10
               b(l) = b(k)
               b(k) = t
   10       continue
            call daxpy(n - k, t, a(k + 1,k), 1, b(k + 1), 1)
   20    continue
   30    continue
c
c        now solve  u*x = y
```

```
c
         do 40 kb = 1, n
            k = n + 1 - kb
            b(k) = b(k) / a(k,k)
            t = -b(k)
            call daxpy(k - 1, t, a(1,k), 1, b(1), 1)
   40    continue
      go to 100
   50 continue
c
c        job = nonzero, solve  trans(a) * x = b
c        first solve  trans(u)*y = b
c
         do 60 k = 1, n
            t = ddot(k - 1, a(1,k), 1, b(1), 1)
            b(k) = (b(k) - t) / a(k,k)
   60    continue
c
c        now solve trans(l)*x = y
c
         if (nm1 .lt. 1) go to 90
         do 80 kb = 1, nm1
            k = n - kb
            b(k) = b(k) + ddot(n - k, a(k + 1,k), 1, b(k + 1), 1)
            l = ipvt(k)
            if (l .eq. k) go to 70
               t = b(l)
               b(l) = b(k)
               b(k) = t
   70       continue
   80    continue
   90    continue
  100 continue
      return
      end

C     ----------------------------------------------------------------

      subroutine daxpy(n, da, dx, incx, dy, incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loops for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      implicit none
c
      integer i, incx, incy, ix, iy, m, mp1
      integer*4 n
      double precision dx(1), dy(1), da
c
      if (n .le. 0) return
      if (da .eq. 0.0d0) return
      if (incx .eq. 1 .and. incy .eq. 1) go to 20
c
c        code for unequal increments or equal increments
c        not equal to 1
c
      ix = 1
      iy = 1
      if (incx .lt. 0) ix = (-n + 1) * incx + 1
      if (incy .lt. 0) iy = (-n + 1) * incy + 1
      do 10 i = 1, n
        dy(iy) = dy(iy) + da * dx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n, 4)
      if ( m .eq. 0 ) go to 40
      do 30 i = 1, m
        dy(i) = dy(i) + da * dx(i)
   30 continue
      if ( n .lt. 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1, n, 4
        dy(i) = dy(i) + da * dx(i)
        dy(i + 1) = dy(i + 1) + da * dx(i + 1)
        dy(i + 2) = dy(i + 2) + da * dx(i + 2)
        dy(i + 3) = dy(i + 3) + da * dx(i + 3)
   50 continue
      return
      end
C     ----------------------------------------------------------------

      subroutine dscal(n, da, dx, incx)
c
c     scales a vector by a constant.
c     uses unrolled loops for increment equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      implicit none
c
      integer i, incx, m, mp1, nincx
      integer*4 n
      double precision da, dx(1)
c
      if (n.le.0) return
      if (incx .eq. 1) go to 20
c
c        code for increment not equal to 1
c
      nincx = n * incx
      do 10 i = 1, nincx, incx
        dx(i) = da * dx(i)
   10 continue
      return
c
c        code for increment equal to 1
c
c
c        clean-up loop
c
```

```
   20 m = mod(n, 5)
      if ( m .eq. 0 ) go to 40
      do 30 i = 1, m
        dx(i) = da * dx(i)
   30 continue
      if ( n .lt. 5 ) return
   40 mp1 = m + 1
      do 50 i = mp1, n, 5
        dx(i) = da * dx(i)
        dx(i + 1) = da * dx(i + 1)
        dx(i + 2) = da * dx(i + 2)
        dx(i + 3) = da * dx(i + 3)
        dx(i + 4) = da * dx(i + 4)
   50 continue
      return
      end

C     ----------------------------------------------------------------

      double precision function ddot(n, dx, incx, dy, incy)
c
c     forms the dot product of two vectors.
c     uses unrolled loops for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      implicit none
c
      integer i, incx, incy, ix, iy, m, mp1
      integer*4 n
      double precision dx(1), dy(1), dtemp
c
      ddot = 0.0d0
      dtemp = 0.0d0
      if (n .le. 0) return
      if (incx .eq. 1 .and. incy .eq. 1) go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if (incx .lt. 0) ix = (-n + 1) * incx + 1
      if (incy .lt. 0) iy = (-n + 1) * incy + 1
      do 10 i = 1, n
        dtemp = dtemp + dx(ix) * dy(iy)
        ix = ix + incx
        iy = iy + incy
   10 continue
      ddot = dtemp
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n, 5)
      if ( m .eq. 0 ) go to 40
      do 30 i = 1,m
```

```
         dtemp = dtemp + dx(i) * dy(i)
   30 continue
      if ( n .lt. 5 ) go to 60
   40 mp1 = m + 1
      do 50 i = mp1, n, 5
         dtemp = dtemp + dx(i) * dy(i) + dx(i + 1) * dy(i + 1) +
     *           dx(i + 2) * dy(i + 2) + dx(i + 3) * dy(i + 3) +
     *           dx(i + 4) * dy(i + 4)
   50 continue
   60 ddot = dtemp
      return
      end

C     ----------------------------------------------------------------

      integer function idamax(n, dx, incx)
c
c     finds the index of element having max. absolute value.
c     jack dongarra, linpack, 3/11/78.
c
      implicit none
c
      integer i, incx, ix
      integer*4 n
      double precision dx(1), dmax
c
      idamax = 0
      if (n .lt. 1) return
      idamax = 1
      if (n .eq. 1) return
      if (incx .eq. 1) go to 20
c
c        code for increment not equal to 1
c
      ix = 1
      dmax = abs(dx(1))
      ix = ix + incx
      do 10 i = 2, n
         if (abs(dx(ix)) .le. dmax) go to 5
         idamax = i
         dmax = abs(dx(ix))
    5    ix = ix + incx
   10 continue
      return
c
c        code for increment equal to 1
c
   20 dmax = abs(dx(1))
      do 30 i = 2, n
         if (abs(dx(i)) .le. dmax) go to 30
         idamax = i
         dmax = abs(dx(i))
   30 continue
      return
      end

C     ----------------------------------------------------------------

      subroutine  dcopy(n, dx, incx, dy, incy)
```

```
c
c     copies a vector, x, to a vector, y.
c     uses unrolled loops for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      implicit none
c
      integer i, incx, incy, ix, iy, m, mp1
      integer*4 n
      double precision dx(1), dy(1)
c
      if (n .le. 0) return
      if (incx .eq. 1 .and. incy .eq. 1) go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if (incx .lt. 0) ix = (-n + 1) * incx + 1
      if (incy .lt. 0) iy = (-n + 1) * incy + 1
      do 10 i = 1, n
        dy(iy) = dx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n, 7)
      if ( m .eq. 0 ) go to 40
      do 30 i = 1, m
        dy(i) = dx(i)
   30 continue
      if ( n .lt. 7 ) return
   40 mp1 = m + 1
      do 50 i = mp1, n, 7
        dy(i) = dx(i)
        dy(i + 1) = dx(i + 1)
        dy(i + 2) = dx(i + 2)
        dy(i + 3) = dx(i + 3)
        dy(i + 4) = dx(i + 4)
        dy(i + 5) = dx(i + 5)
        dy(i + 6) = dx(i + 6)
   50 continue
      return
      end
```

H Listing of fcvkryx_bbd_p.f

```
C     ----------------------------------------------------------------
C     $Revision: 1.1 $
C     $Date: 2006/07/05 15:50:04 $
C     ----------------------------------------------------------------
C     Diagonal ODE example.  Stiff case, with diagonal preconditioner.
C     Uses FCVODE interfaces and FCVBBD interfaces.
C     Solves problem twice -- with left and right preconditioning.
C     ----------------------------------------------------------------
C
C     Include MPI-Fortran header file for MPI_COMM_WORLD, MPI types.

      IMPLICIT NONE
C
      INCLUDE "mpif.h"
C
      INTEGER*4 NLOCAL
      PARAMETER (NLOCAL=10)
C
      INTEGER NOUT, LNST, LNFE, LNSETUP, LNNI, LNCF, LNETF, LNPE
      INTEGER LNLI, LNPS, LNCFL, MYPE, IER, NPES, METH, ITMETH
      INTEGER LLENRW, LLENIW, LLENRWLS, LLENIWLS
      INTEGER IATOL, ITASK, IPRE, IGS, JOUT
      INTEGER*4 IOUT(25), IPAR(2)
      INTEGER*4 NEQ, I, MUDQ, MLDQ, MU, ML, NETF
      INTEGER*4 NST, NFE, NPSET, NPE, NPS, NNI, NLI, NCFN, NCFL, NGEBBD
      INTEGER*4 LENRW, LENIW, LENRWLS, LENIWLS, LENRWBBD, LENIWBBD
      DOUBLE PRECISION Y(1024), ROUT(10), RPAR(1)
      DOUBLE PRECISION ALPHA, TOUT, ERMAX, AVDIM
      DOUBLE PRECISION ATOL, ERRI, RTOL, GERMAX, DTOUT, T
C
      DATA ATOL/1.0D-10/, RTOL/1.0D-5/, DTOUT/0.1D0/, NOUT/10/
      DATA LLENRW/1/, LLENIW/2/, LNST/3/, LNFE/4/, LNETF/5/,  LNCF/6/,
     1     LNNI/7/, LNSETUP/8/, LLENRWLS/13/, LLENIWLS/14/,
     1     LNPE/18/, LNLI/20/, LNPS/19/, LNCFL/21/
C
C     Get NPES and MYPE.  Requires initialization of MPI.
      CALL MPI_INIT(IER)
      IF (IER .NE. 0) THEN
         WRITE(6,5) IER
 5       FORMAT(///' MPI_ERROR: MPI_INIT returned IER = ', I5)
         STOP
      ENDIF
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,6) IER
 6       FORMAT(///' MPI_ERROR: MPI_COMM_SIZE returned IER = ', I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYPE, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,7) IER
 7       FORMAT(///' MPI_ERROR: MPI_COMM_RANK returned IER = ', I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
C
```

```
C      Set input arguments.
       NEQ = NPES * NLOCAL
       T = 0.0D0
       METH = 2
       ITMETH = 2
       IATOL = 1
       ITASK = 1
       IPRE = 1
       IGS = 1
C      Set parameter alpha
       ALPHA  = 10.0D0
C
C      Load IPAR and RPAR
       IPAR(1) = NLOCAL
       IPAR(2) = MYPE
       RPAR(1) = ALPHA
C
       DO I = 1, NLOCAL
          Y(I) = 1.0D0
       ENDDO
C
       IF (MYPE .EQ. 0) THEN
          WRITE(6,15) NEQ, ALPHA, RTOL, ATOL, NPES
 15       FORMAT('Diagonal test problem:'//' NEQ = ', I3, /
     &           ' parameter alpha = ', F8.3/
     &           ' ydot_i = -alpha*i * y_i (i = 1,...,NEQ)'/
     &           ' RTOL, ATOL = ', 2E10.1/
     &           ' Method is BDF/NEWTON/SPGMR'/
     &           ' Preconditioner is band-block-diagonal, using CVBBDPRE'
     &           /' Number of processors = ', I3/)
       ENDIF
C
       CALL FNVINITP(MPI_COMM_WORLD, 1, NLOCAL, NEQ, IER)
C
       IF (IER .NE. 0) THEN
          WRITE(6,20) IER
 20       FORMAT(///' SUNDIALS_ERROR: FNVINITP returned IER = ', I5)
          CALL MPI_FINALIZE(IER)
          STOP
       ENDIF
C
       CALL FCVMALLOC(T, Y, METH, ITMETH, IATOL, RTOL, ATOL,
     &                IOUT, ROUT, IPAR, RPAR, IER)
C
       IF (IER .NE. 0) THEN
          WRITE(6,30) IER
 30       FORMAT(///' SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
          CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
          STOP
       ENDIF
C
       MUDQ = 0
       MLDQ = 0
       MU = 0
       ML = 0
       CALL FCVBBDINIT(NLOCAL, MUDQ, MLDQ, MU, ML, 0.0D0, IER)
       IF (IER .NE. 0) THEN
          WRITE(6,35) IER
 35       FORMAT(///' SUNDIALS_ERROR: FCVBBDINIT returned IER = ', I5)
```

```
            CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
            STOP
         ENDIF
C
         CALL FCVBBDSPGMR(IPRE, IGS, 0, 0.0D0, IER)
         IF (IER .NE. 0) THEN
            WRITE(6,36) IER
 36         FORMAT(///' SUNDIALS_ERROR: FCVBBDSPGMR returned IER = ', I5)
            CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
            STOP
         ENDIF
C
         IF (MYPE .EQ. 0) WRITE(6,38)
 38      FORMAT(/'Preconditioning on left'/)
C
C     Looping point for cases IPRE = 1 and 2.
C
 40      CONTINUE
C
C     Loop through tout values, call solver, print output, test for failure.
         TOUT = DTOUT
         DO 60 JOUT = 1, NOUT
C
            CALL FCVODE(TOUT, T, Y, ITASK, IER)
C
            IF (MYPE .EQ. 0) WRITE(6,45) T, IOUT(LNST), IOUT(LNFE)
 45         FORMAT(' t = ', E10.2, 5X, 'no. steps = ', I5,
     &             '   no. f-s = ', I5)
C
            IF (IER .NE. 0) THEN
               WRITE(6,50) IER, IOUT(15)
 50            FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
     &                '                 Linear Solver returned IER = ', I5)
               CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
               STOP
            ENDIF
C
            TOUT = TOUT + DTOUT
 60      CONTINUE
C
C     Get max. absolute error in the local vector.
         ERMAX = 0.0D0
         DO 65 I = 1, NLOCAL
            ERRI  = Y(I) - EXP(-ALPHA * (MYPE * NLOCAL + I) * T)
            ERMAX = MAX(ERMAX, ABS(ERRI))
 65      CONTINUE
C     Get global max. error from MPI_REDUCE call.
         CALL MPI_REDUCE(ERMAX, GERMAX, 1, MPI_DOUBLE_PRECISION, MPI_MAX,
     &                   0, MPI_COMM_WORLD, IER)
         IF (IER .NE. 0) THEN
            WRITE(6,70) IER
 70         FORMAT(///' MPI_ERROR: MPI_REDUCE returned IER = ', I5)
            CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
            STOP
         ENDIF
         IF (MYPE .EQ. 0) WRITE(6,75) GERMAX
 75      FORMAT(/'Max. absolute error is', E10.2/)
C
C     Print final statistics.
```

```
      IF (MYPE .EQ. 0) THEN
         NST = IOUT(LNST)
         NFE = IOUT(LNFE)
         NPSET = IOUT(LNSETUP)
         NPE = IOUT(LNPE)
         NPS = IOUT(LNPS)
         NNI = IOUT(LNNI)
         NLI = IOUT(LNLI)
         AVDIM = DBLE(NLI) / DBLE(NNI)
         NCFN = IOUT(LNCF)
         NCFL = IOUT(LNCFL)
         NETF = IOUT(LNETF)
         LENRW = IOUT(LLENRW)
         LENIW = IOUT(LLENIW)
         LENRWLS = IOUT(LLENRWLS)
         LENIWLS = IOUT(LLENIWLS)
         WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
     &               NCFL, NETF, LENRW, LENIW, LENRWLS, LENIWLS
 80      FORMAT(/'Final statistics:'//
     &          ' number of steps        = ', I5, 4X,
     &          ' number of f evals.     = ', I5/
     &          ' number of prec. setups = ', I5/
     &          ' number of prec. evals. = ', I5, 4X,
     &          ' number of prec. solves = ', I5/
     &          ' number of nonl. iters. = ', I5, 4X,
     &          ' number of lin. iters.  = ', I5/
     &          ' average Krylov subspace dimension (NLI/NNI) = ',F8.4/
     &          ' number of conv. failures.. nonlinear = ', I3,
     &          '  linear = ', I3/
     &          ' number of error test failures = ', I3/
     &          ' main solver real/int workspace sizes   = ',2I5/
     &          ' linear solver real/int workspace sizes = ',2I5)
         CALL FCVBBDOPT(LENRWBBD, LENIWBBD, NGEBBD)
         WRITE(6,82) LENRWBBD, LENIWBBD, NGEBBD
 82      FORMAT('In CVBBDPRE:'/
     &          ' real/int local workspace = ', 2I5/
     &          ' number of g evals. = ', I5)
      ENDIF
C
C     If IPRE = 1, re-initialize T, Y, and the solver, and loop for
C     case IPRE = 2.  Otherwise jump to final block.
      IF (IPRE .EQ. 2) GO TO 99
C
      T = 0.0D0
      DO I = 1, NLOCAL
         Y(I) = 1.0D0
      ENDDO
C
      CALL FCVREINIT(T, Y, IATOL, RTOL, ATOL, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,91) IER
 91      FORMAT(///' SUNDIALS_ERROR: FCVREINIT returned IER = ', I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
C
      IPRE = 2
C
      CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, 0.0D0, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,92) IER
 92      FORMAT(///' SUNDIALS_ERROR: FCVBBDREINIT returned IER = ', I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
C
      CALL FCVSPGMRREINIT(IPRE, IGS, 0.0D0, IER)
      IF (IER .NE. 0) THEN
         WRITE(6,93) IER
 93      FORMAT(///' SUNDIALS_ERROR: FCVSPGMRREINIT returned IER = ',I5)
         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
         STOP
      ENDIF
C
      IF (MYPE .EQ. 0) WRITE(6,95)
 95   FORMAT(//60('-')///'Preconditioning on right'/)
      GO TO 40
C
C     Free the memory and finalize MPI.
 99   CALL FCVBBDFREE
      CALL FCVFREE
      CALL MPI_FINALIZE(IER)
C
      STOP
      END
C
C     ------------------------------------------------------------------------
C
      SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
C     Routine for right-hand side function f
      IMPLICIT NONE
C
      INTEGER*4 IPAR(*), IER
      DOUBLE PRECISION T, Y(*), YDOT(*), RPAR(*)
C
      INTEGER MYPE
      INTEGER*4 I, NLOCAL
      DOUBLE PRECISION ALPHA
C
      NLOCAL = IPAR(1)
      MYPE = IPAR(2)
      ALPHA = RPAR(1)
C
      DO I = 1, NLOCAL
         YDOT(I) = -ALPHA * (MYPE * NLOCAL + I) * Y(I)
      ENDDO
C
      IER = 0
C
      RETURN
      END
C
C     ------------------------------------------------------------------------
C
      SUBROUTINE FCVGLOCFN(NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
C     Routine to define local approximate function g, here the same as f.
      IMPLICIT NONE
C
```

```
      INTEGER*4 NLOC, IPAR(*), IER
      DOUBLE PRECISION T, YLOC(*), GLOC(*), RPAR(*)
C
      CALL FCVFUN(T, YLOC, GLOC, IPAR, RPAR, IER)
C
      RETURN
      END
C
C     ------------------------------------------------------------------------
C
      SUBROUTINE FCVCOMMFN(NLOC, T, YLOC, IPAR, RPAR, IER)
C     Routine to perform communication required for evaluation of g.
      IER = 0
      RETURN
      END
```