UCRL-SM-208110

Example Programs for CVODE v2.2.0

Alan C. Hindmarsh and Radu Serban

November 2004

DISCLAIMER

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.

Example Programs for CVODE v2.2.0

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

Contents

1 Introduction **1**

2 Serial example problems **4**
2.1 A dense example: `cvdx` 4
2.2 A banded example: `cvbx` 6
2.3 A Krylov example: `cvkx` 9

3 Parallel example problems **13**
3.1 A nonstiff example: `pvnx` 13
3.2 A user preconditioner example: `pvkx` 15
3.3 A CVBBDPRE preconditioner example: `pvkxb` 17

4 Fortran example problems **21**
4.1 A serial example: `cvkryf` 21
4.2 A parallel example: `pvdiagkbf` 23

5 Parallel tests **25**

References **27**

A Listing of `cvdx.c` **28**

B Listing of `cvbx.c` **35**

C Listing of `cvkx.c` **44**

D Listing of `pvnx.c` **57**

E Listing of `pvkx.c` **64**

F Listing of `pvkxb.c` **83**

G Listing of `cvkryf.f` **100**

H Listing of `pvdiagkbf.f` **116**

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

- `cvdx` 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.

- `cvbx` 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.

- `cvkx` 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.

- `cvkxb` solves the same problem as `cvkx`, 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.

- `cvdemd` 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.

- `cvdemk` 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/cvode/examples_par` directory are the following three parallel examples (using the NVECTOR_PARALLEL module):

- `pvnx` 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.

- `pvkx` is the parallel implementation of `cvkx`.

- `pvkxb` solves the same problem as `pvkx`, 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/cvode/fmix/examples_ser` and `sundials/cvode/fmix/examples_par`, are the following examples for the FORTRAN-C interface:

- `cvdensef` is a serial chemical kinetics example (`BDF`/DENSE) with rootfinding.

- `cvbandf` is a serial advection-diffusion example (`BDF`/BAND).

- `cvkryf` is a serial kinetics-transport example (`BDF`/SPGMR).

- `cvkrybf` is the `cvkryf` example with FCVBP.

- `pvdiagnf` is a parallel diagonal ODE example (`ADAMS`/`FUNCTIONAL`).

- `pvdiagkf` is a parallel diagonal ODE example (`BDF`/SPGMR).

- `pvdiagkbf` 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 `cvkx`/`pvkx` 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: cvdx

As an initial illustration of the use of the CVODE package for the integration of IVP ODEs, we give a sample program called `cvdx.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 `sundialstypes.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 `cvdense.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. Finally, the `dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. We have explicitly included `dense.h`, but this is not necessary because it is included by `cvdense.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 §6.1. The macro `DENSE_ELEM` is fully described in §5.6.2.

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 two 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 set the absolute tolerance vector 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 address 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 §5.5.2) and CVDenseSetJacFn (see §5.5.4) 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 §5.5.3 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 §5.5.6.

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 §5.6.2 for a detailed description of the dense `Jac` function.

The output generated by `cvdx` 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.

cvdx 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  = 754    nsetups = 110    nfeD = 0      njeD = 12
nni = 751    ncfn = 0      netf = 26     nge = 541
```

2.2 A banded example: `cvbx`

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

The cvbx.c program includes files cvband.h and band.h in order to use the CVBAND linear solver. The cvband.h file contains the prototype for the CVBand routine. The 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 §8.2). We have explicitly included band.h, but this is not necessary because it is included by cvband.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 CVBandSetJacData, 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 §5.5.2) specifies the CVBAND linear solver, and specifies that both half-bandwidths of the Jacobian are equal to MY. The call to CVBandSetJacFn (see §5.5.4) specifies that a user-supplied Jacobian function Jac is to be used. The actual solution of the problem is performed by the call to CVode within the loop over the output times tout. The max-norm of the solution vector (from a call to N_VMaxNorm) and the cumulative number of time steps (from a call to CVodeGetNumSteps) are printed at each output time. Finally, the calls to PrintFinalStats, N_VDestroy, and CVodeFree print statistics and free problem memory.

Following the main program in the cvbx.c file are definitions of five functions: f, Jac, SetIC, PrintFinalStats, and check_flag. The last three functions are called only from within the cvbx.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 §5.5.6. 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 §5.6.3 for a detailed description of the banded `Jac` function.

The output generated by `cvbx` is shown below.

cvbx 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  = 173    nsetups = 23     nfeB = 0      njeB = 3
nni = 170    ncfn = 0      netf = 3
```

2.3 A Krylov example: cvkx

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, `cvkx.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

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

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

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

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

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

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

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

Among the initial `#include` lines in this case are lines to include `cvspgmr.h` and `sundialsmath.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 `sundialsmath.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 §5.5.2) 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 `CVSpgmrSetGSType`. Next, user-supplied preconditioner setup and solve functions, `Precond` and `PSolve`, are specified through calls to `CVSpgmrSetPrecSetupFn` and `CVSpgmrSetPrecSolveFn`, respectively. The `data` pointer

passed to CVSpgmrSetPrecData is passed to Precond and PSolve whenever these are called. See §5.5.4 for details on these CVSpgmrSet* functions.

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 §5.5.6. 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 §5.6.6) and backsolved in the PSolve routine (see §5.6.5). 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 cvkx.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, cvkx.c uses the small dense functions for all operations on the 2×2 preconditioner blocks. Thus it includes smalldense.h, and calls the small dense matrix functions denalloc, dencopy, denscale, denaddI, denfree, denfreepiv, gefa, and gesl. 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 §8.1.

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

cvkx 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 = 310   order = 4   stepsize = 1.80e+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 = 340   order = 5   stepsize = 1.15e+02
c1 (bot.left/middle/top rt.) =    1.404e+04     2.029e+04     1.561e+04
c2 (bot.left/middle/top rt.) =    3.387e+11     4.894e+11     3.765e+11

t = 4.32e+04   no. steps = 393   order = 5   stepsize = 7.13e+02
c1 (bot.left/middle/top rt.) =   -4.343e-06    -1.125e-04    -5.057e-06
c2 (bot.left/middle/top rt.) =    3.382e+11     1.355e+11     3.804e+11

t = 5.04e+04   no. steps = 404   order = 5   stepsize = 7.37e+02
c1 (bot.left/middle/top rt.) =    1.055e-07     1.307e-06     5.090e-08
c2 (bot.left/middle/top rt.) =    3.358e+11     4.930e+11     3.864e+11

t = 5.76e+04   no. steps = 415   order = 5   stepsize = 4.24e+02
c1 (bot.left/middle/top rt.) =   -1.727e-10    -5.322e-07     2.131e-09
c2 (bot.left/middle/top rt.) =    3.320e+11     9.650e+11     3.909e+11

t = 6.48e+04   no. steps = 429   order = 5   stepsize = 8.83e+02
c1 (bot.left/middle/top rt.) =    6.386e-09     6.542e-05    -1.998e-07
c2 (bot.left/middle/top rt.) =    3.313e+11     8.922e+11     3.963e+11

t = 7.20e+04   no. steps = 437   order = 5   stepsize = 8.83e+02
c1 (bot.left/middle/top rt.) =    2.970e-09     3.114e-05    -9.487e-08
c2 (bot.left/middle/top rt.) =    3.330e+11     6.186e+11     4.039e+11

t = 7.92e+04   no. steps = 445   order = 5   stepsize = 8.83e+02
c1 (bot.left/middle/top rt.) =   -8.014e-12    -1.553e-09     3.265e-11
```

```
c2 (bot.left/middle/top rt.) =    3.334e+11     6.669e+11     4.120e+11

t = 8.64e+04   no. steps = 453   order = 5   stepsize = 8.83e+02
c1 (bot.left/middle/top rt.) =   -8.955e-11    -9.252e-07     2.824e-09
c2 (bot.left/middle/top rt.) =    3.352e+11     9.106e+11     4.162e+11


Final Statistics..

lenrw   =  2000     leniw =    10
llrw    =  2046     lliw  =    10
nst     =   453
nfe     =   583     nfel  =   615
nni     =   580     nli   =   615
nsetups =    76     netf  =    25
npe     =     8     nps   =  1142
ncfn    =     0     ncfl  =     1
```

3 Parallel example problems

3.1 A nonstiff example: pvnx

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

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

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

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

A system of MX ODEs is obtained by discretizing the x-axis with MX+2 grid points and replacing the first and second order spatial derivatives with their central difference approximations. Since the value of u is constant at the two endpoints, the semi-discrete equations for those points can be eliminated. With u_i as the approximation to $u(t, x_i)$, $x_i = i(\Delta x)$, and $\Delta x = 2/(\mathtt{MX}+1)$, 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 $u_0 = u_{\mathtt{MX}+1} = 0$.

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, pvnx.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 pvnx.c file begins with #include lines, which include 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 `pvnx.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 `pvnx` 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.

pvnx 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) =  5.772861e-03  nst = 481
At t = 3.50  max.norm(u) =  1.650209e-03  nst = 551
At t = 4.00  max.norm(u) =  4.718756e-04  nst = 622
At t = 4.50  max.norm(u) =  1.360229e-04  nst = 695
At t = 5.00  max.norm(u) =  4.044654e-05  nst = 761

Final Statistics:

nst = 761     nfe  = 1380    nni = 0       ncfn = 128     netf = 5
```

3.2 A user preconditioner example: pvkx

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 `cvkx` 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 `pvnx`, 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 `pvkx` 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 `pvkx.c` are the same as in `cvkx.c`.

The following is a sample output from `pvkx`, 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.

pvkx 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 = 308   order = 4   stepsize = 2.40e+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 = 345   order = 4   stepsize = 7.45e+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 = 402   order = 5   stepsize = 3.33e+02
At bottom left:  c1, c2 =    2.995e-06    3.382e+11
At top right:    c1, c2 =    3.316e-06    3.804e+11

t = 5.04e+04   no. steps = 423   order = 5   stepsize = 3.33e+02
At bottom left:  c1, c2 =   -3.405e-06    3.358e+11
At top right:    c1, c2 =   -3.879e-06    3.864e+11

t = 5.76e+04   no. steps = 435   order = 5   stepsize = 5.93e+02
At bottom left:  c1, c2 =   -6.650e-07    3.320e+11
At top right:    c1, c2 =   -7.564e-07    3.909e+11

t = 6.48e+04   no. steps = 448   order = 5   stepsize = 6.15e+02
At bottom left:  c1, c2 =   -5.322e-07    3.313e+11
At top right:    c1, c2 =   -6.039e-07    3.963e+11

t = 7.20e+04   no. steps = 460   order = 5   stepsize = 6.15e+02
At bottom left:  c1, c2 =   -2.992e-07    3.330e+11
At top right:    c1, c2 =   -3.395e-07    4.039e+11

t = 7.92e+04   no. steps = 471   order = 5   stepsize = 6.15e+02
At bottom left:  c1, c2 =    1.582e-08    3.334e+11
At top right:    c1, c2 =    1.800e-08    4.120e+11

t = 8.64e+04   no. steps = 483   order = 5   stepsize = 6.15e+02
At bottom left:  c1, c2 =   -9.920e-11    3.352e+11
At top right:    c1, c2 =   -1.133e-10    4.163e+11


Final Statistics:

lenrw   =  2000     leniw =    80
llrw    =  2046     lliw  =    80
nst     =   483
nfe     =   615     nfel  =   609
nni     =   612     nli   =   609
nsetups =    77     netf  =    26
npe     =     9     nps   =  1170
ncfn    =     0     ncfl  =     0
```

3.3 A CVBBDPRE preconditioner example: pvkxb

In this example, pvkxb, we solve the same problem in pvkx 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 pvkx.c, the f routine in pvkxb.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 pvkxb.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 (CVSpgmrSetPrecType).

Sample output from pvkxb 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 pvkx when preconditioning is on the left, but slightly poorer when it is on the right.

pvkxb 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 = 3.01e+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 = 268   order = 5   stepsize = 1.42e+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 = 314   order = 4   stepsize = 8.08e+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 = 386   order = 4   stepsize = 4.22e+02
At bottom left:  c1, c2 =   -2.127e-07    3.382e+11
At top right:    c1, c2 =    3.231e-08    3.804e+11

t = 5.04e+04   no. steps = 399   order = 5   stepsize = 4.74e+02
At bottom left:  c1, c2 =    3.071e-09    3.358e+11
At top right:    c1, c2 =    8.013e-08    3.864e+11

t = 5.76e+04   no. steps = 411   order = 5   stepsize = 4.15e+02
At bottom left:  c1, c2 =    7.177e-11    3.320e+11
At top right:    c1, c2 =    1.449e-09    3.909e+11

t = 6.48e+04   no. steps = 422   order = 5   stepsize = 6.69e+02
At bottom left:  c1, c2 =   -4.098e-12    3.313e+11
At top right:    c1, c2 =   -8.238e-11    3.963e+11

t = 7.20e+04   no. steps = 433   order = 5   stepsize = 6.69e+02
At bottom left:  c1, c2 =    6.565e-14    3.330e+11
At top right:    c1, c2 =    1.339e-12    4.039e+11

t = 7.92e+04   no. steps = 444   order = 5   stepsize = 6.69e+02
At bottom left:  c1, c2 =   -2.015e-15    3.334e+11
At top right:    c1, c2 =   -5.290e-14    4.120e+11

t = 8.64e+04   no. steps = 454   order = 5   stepsize = 6.69e+02
At bottom left:  c1, c2 =   -1.686e-17    3.352e+11
At top right:    c1, c2 =   -2.079e-16    4.163e+11


Final Statistics:

lenrw   =  2000     leniw =    80
llrw    =  2046     lliw  =    80
nst     =   454
nfe     =   592     nfel  =   558
nni     =   589     nli   =   558
nsetups =    80     netf  =    30
npe     =     9     nps   =  1101
ncfn    =     0     ncfl  =     0

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


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


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 = 322   order = 3   stepsize = 1.09e+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 = 361   order = 5   stepsize = 7.66e+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 = 415   order = 5   stepsize = 5.93e+02
At bottom left:  c1, c2 =    7.315e-09    3.382e+11
At top right:    c1, c2 =    8.641e-09    3.804e+11

t = 5.04e+04   no. steps = 428   order = 5   stepsize = 6.79e+02
At bottom left:  c1, c2 =   -2.165e-11    3.358e+11
At top right:    c1, c2 =    4.943e-11    3.864e+11

t = 5.76e+04   no. steps = 442   order = 4   stepsize = 2.01e+02
At bottom left:  c1, c2 =    2.578e-09    3.320e+11
At top right:    c1, c2 =    1.422e-08    3.909e+11

t = 6.48e+04   no. steps = 457   order = 4   stepsize = 5.51e+02
At bottom left:  c1, c2 =   -7.868e-12    3.313e+11
At top right:    c1, c2 =    1.272e-11    3.963e+11

t = 7.20e+04   no. steps = 470   order = 4   stepsize = 5.51e+02
At bottom left:  c1, c2 =   -4.662e-13    3.330e+11
At top right:    c1, c2 =   -8.547e-13    4.039e+11

t = 7.92e+04   no. steps = 483   order = 4   stepsize = 5.51e+02
At bottom left:  c1, c2 =    1.121e-15    3.334e+11
At top right:    c1, c2 =    6.400e-16    4.120e+11

t = 8.64e+04   no. steps = 496   order = 4   stepsize = 5.51e+02
At bottom left:  c1, c2 =    7.299e-16    3.352e+11
At top right:    c1, c2 =   -3.965e-19    4.163e+11


Final Statistics:

lenrw   =  2000     leniw =    80
llrw    =  2046     lliw  =    80
nst     =   496
```

```
nfe     =   652     nfel  =   852
nni     =   649     nli   =   852
nsetups =   104     netf  =    35
npe     =     9     nps   =  1381
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 F77 Fortran 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: `cvkryf`

The `cvkryf` example is a Fortran equivalent of the `cvkx` problem. (In fact, it was derived from an earlier Fortran example program for VODPK.) The source program `cvkryf.c` is listed in Appendix G.

The main program begins with a call to `INITKX`, which sets problem parameters, loads these in a Common block 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 `IOPT` and `ROPT` arrays). At the end, it prints a number of performance counters, and frees memory with calls to `FCVFREE` and `FNVFREES`.

In `cvkryf.c`, 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 `JOK` $= 0$, but is simply copied from `BD` to `P` if `JOK` $= 1$. In both cases, the preconditioner matrix P is formed from J_{bd} and its 2×2 blocks are LU-factored. In `FCVPSOL`, the solution of a linear system $Px = z$ is solved by doing backsolve operations on the blocks. The remainder of `cvkryf.c` consists of routines from LINPACK and the BLAS needed for matrix and vector operations.

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

cvkryf sample output

```
Krylov example problem:

 Kinetics-transport, NEQ =  200


 t =   0.720E+04     no. steps =   219   order =   5   stepsize =   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     no. steps =   251   order =   5   stepsize =   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     no. steps =   277   order =   5   stepsize =   0.274583E+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     no. steps =   307   order =   4   stepsize =   0.198295E+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     no. steps =   338   order =   5   stepsize =   0.115649E+03
  c1 (bot.left/middle/top rt.) =   0.140404E+05  0.202903E+05  0.156090E+05
  c2 (bot.left/middle/top rt.) =   0.338677E+12  0.489443E+12  0.376516E+12

 t =   0.432E+05     no. steps =   391   order =   5   stepsize =   0.541755E+03
  c1 (bot.left/middle/top rt.) =   0.303486E-06 -0.149368E-04  0.358694E-06
  c2 (bot.left/middle/top rt.) =   0.338233E+12  0.135488E+12  0.380352E+12

 t =   0.504E+05     no. steps =   406   order =   5   stepsize =   0.333083E+03
  c1 (bot.left/middle/top rt.) =   0.118059E-06  0.293305E-04  0.520748E-06
  c2 (bot.left/middle/top rt.) =   0.335816E+12  0.493020E+12  0.386445E+12

 t =   0.576E+05     no. steps =   426   order =   4   stepsize =   0.524527E+03
  c1 (bot.left/middle/top rt.) =  -0.131214E-09 -0.296505E-05  0.119711E-07
  c2 (bot.left/middle/top rt.) =   0.332031E+12  0.964985E+12  0.390900E+12

 t =   0.648E+05     no. steps =   440   order =   4   stepsize =   0.524527E+03
  c1 (bot.left/middle/top rt.) =  -0.107677E-14 -0.679419E-11  0.431755E-13
  c2 (bot.left/middle/top rt.) =   0.331303E+12  0.892179E+12  0.396342E+12

 t =   0.720E+05     no. steps =   454   order =   4   stepsize =   0.524527E+03
  c1 (bot.left/middle/top rt.) =   0.130324E-18  0.339889E-13 -0.502096E-16
  c2 (bot.left/middle/top rt.) =   0.332972E+12  0.618620E+12  0.403885E+12

 t =   0.792E+05     no. steps =   467   order =   4   stepsize =   0.524527E+03
  c1 (bot.left/middle/top rt.) =   0.134675E-18 -0.701612E-13  0.494953E-18
  c2 (bot.left/middle/top rt.) =   0.333441E+12  0.666873E+12  0.412026E+12

 t =   0.864E+05     no. steps =   481   order =   4   stepsize =   0.524527E+03
  c1 (bot.left/middle/top rt.) =   0.229835E-19  0.389838E-13 -0.136143E-17
  c2 (bot.left/middle/top rt.) =   0.335178E+12  0.910760E+12  0.416251E+12


Final statistics:

 number of steps        =   481     number of f evals.     =  608
 number of prec. setups =    75
 number of prec. evals. =     8     number of prec. solves =  1202
 number of nonl. iters. =   605     number of lin. iters.  =   651
 average Krylov subspace dimension (NLI/NNI)  =   0.107603E+01
 number of conv. failures.. nonlinear =   0  linear =   0
```

4.2 A parallel example: pvdiagkbf

This example, pvdiagkbf, 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 $0 \le t \le 1$. 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, pvdiagkbf.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 pvdiagkbf, with NPES = 4. As expected, the performance is identical for left vs right preconditioning.

pvdiagkbf 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 =   261
 t =   0.20E+00     no. steps =   265   no. f-s =   307
 t =   0.30E+00     no. steps =   290   no. f-s =   333
 t =   0.40E+00     no. steps =   306   no. f-s =   350
 t =   0.50E+00     no. steps =   319   no. f-s =   364
 t =   0.60E+00     no. steps =   329   no. f-s =   374
 t =   0.70E+00     no. steps =   339   no. f-s =   385
 t =   0.80E+00     no. steps =   345   no. f-s =   391
 t =   0.90E+00     no. steps =   352   no. f-s =   398
 t =   0.10E+01     no. steps =   359   no. f-s =   405

Max. absolute error is  0.28E-08
```

```
Final statistics:

 number of steps        =   359     number of f evals.     =   405
 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

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 =   261
 t =   0.20E+00     no. steps =   265   no. f-s =   307
 t =   0.30E+00     no. steps =   290   no. f-s =   333
 t =   0.40E+00     no. steps =   306   no. f-s =   350
 t =   0.50E+00     no. steps =   319   no. f-s =   364
 t =   0.60E+00     no. steps =   329   no. f-s =   374
 t =   0.70E+00     no. steps =   339   no. f-s =   385
 t =   0.80E+00     no. steps =   345   no. f-s =   391
 t =   0.90E+00     no. steps =   352   no. f-s =   398
 t =   0.10E+01     no. steps =   359   no. f-s =   405

Max. absolute error is  0.28E-08


Final statistics:

 number of steps        =   359     number of f evals.     =   405
 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

In CVBBDPRE:

 real/int local workspace =    20   10
 number of g evals. =    14
```

5 Parallel tests

The stiff example problem `cvkx` described above, or rather its parallel version `pvkx`, 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 v2.2.0. Technical Report UCRL-SM-208108, LLNL, 2004.

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

A Listing of `cvdx.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.19 $
 * $Date: 2004/11/15 18:56:35 $
 * -----------------------------------------------------------------
 * 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 cvdx.c */

#include "sundialstypes.h"   /* definition of type realtype                */
#include "cvode.h"           /* prototypes for CVode* functions and        */
                             /* constants CV_BDF, CV_NEWTON, CV_SV,        */
                             /* CV_NORMAL, CV_SUCCESS, and CV_ROOT_RETURN  */
#include "cvdense.h"         /* prototype for CVDense                      */
#include "nvector_serial.h"  /* definitions of type N_Vector, macro        */
                             /* NV_Ith_S, and prototypes for N_VNew_Serial */
                             /* and N_VDestroy                             */
#include "dense.h"           /* definition of type DenseMat and macro      */
                             /* DENSE_ELEM                                 */


/* 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
```

```
53     the range [1..NEQ] and NEQ is defined below. The Ith macro is defined
54     using the N_VIth macro in nvector.h. N_VIth numbers the components of
55     a vector starting from 0.
56
57     IJth(A,i,j) references the (i,j)th element of the dense matrix A, where
58     i and j are in the range [1..NEQ]. The IJth macro is defined using the
59     DENSE_ELEM macro in dense.h. DENSE_ELEM numbers rows and columns of a
60     dense matrix starting from 0. */
61
62  #define Ith(v,i)    NV_Ith_S(v,i-1)       /* Ith numbers components 1..NEQ */
63  #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */
64
65
66  /* Problem Constants */
67
68  #define NEQ   3                /* number of equations  */
69  #define Y1    RCONST(1.0)      /* initial y components */
70  #define Y2    RCONST(0.0)
71  #define Y3    RCONST(0.0)
72  #define RTOL  RCONST(1.0e-4)   /* scalar relative tolerance            */
73  #define ATOL1 RCONST(1.0e-8)   /* vector absolute tolerance components */
74  #define ATOL2 RCONST(1.0e-14)
75  #define ATOL3 RCONST(1.0e-6)
76  #define T0    RCONST(0.0)      /* initial time           */
77  #define T1    RCONST(0.4)      /* first output time      */
78  #define TMULT RCONST(10.0)     /* output time factor     */
79  #define NOUT  12               /* number of output times */
80
81
82  /* Functions Called by the Solver */
83
84  static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
85
86  static void g(realtype t, N_Vector y, realtype *gout, void *g_data);
87
88  static void Jac(long int N, DenseMat J, realtype t,
89                  N_Vector y, N_Vector fy, void *jac_data,
90                  N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
91
92  /* Private functions to output results */
93
94  static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3);
95  static void PrintRootInfo(int root_f1, int root_f2);
96
97  /* Private function to print final statistics */
98
99  static void PrintFinalStats(void *cvode_mem);
100
101 /* Private function to check function return values */
102
103 static int check_flag(void *flagvalue, char *funcname, int opt);
104
105
106 /*
```

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

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

  y = abstol = NULL;
  cvode_mem = NULL;

  /* Create serial vectors 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
```

```
161       &reltol   is a pointer to the scalar relative tolerance
162       abstol    is the absolute tolerance vector
163    */
164
165    flag = CVodeMalloc(cvode_mem, f, T0, y, CV_SV, &reltol, abstol);
166    if (check_flag(&flag, "CVodeMalloc", 1)) return(1);
167
168    /* Call CVodeRootInit to specify the root function g with 2 components */
169    flag = CVodeRootInit(cvode_mem, g, 2);
170    if (check_flag(&flag, "CVodeRootInit", 1)) return(1);
171
172    /* Call CVDense to specify the CVDENSE dense linear solver */
173    flag = CVDense(cvode_mem, NEQ);
174    if (check_flag(&flag, "CVDense", 1)) return(1);
175
176    /* Set the Jacobian routine to Jac (user-supplied) */
177    flag = CVDenseSetJacFn(cvode_mem, Jac);
178    if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
179
180    /* In loop, call CVode, print results, and test for error.
181       Break out of loop when NOUT preset output times have been reached.  */
182    printf(" \n3-species kinetics problem\n\n");
183
184    iout = 0;  tout = T1;
185    while(1) {
186      flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
187      PrintOutput(t, Ith(y,1), Ith(y,2), Ith(y,3));
188
189      if (flag == CV_ROOT_RETURN) {
190        flagr = CVodeGetRootInfo(cvode_mem, &rootsfound);
191        check_flag(&flagr, "CVodeGetRootInfo", 1);
192        PrintRootInfo(rootsfound[0],rootsfound[1]);
193      }
194
195      if (check_flag(&flag, "CVode", 1)) break;
196      if (flag == CV_SUCCESS) {
197        iout++;
198        tout *= TMULT;
199      }
200
201      if (iout == NOUT) break;
202    }
203
204    /* Print some final statistics */
205    PrintFinalStats(cvode_mem);
206
207    /* Free y and abstol vectors */
208    N_VDestroy_Serial(y);
209    N_VDestroy_Serial(abstol);
210    /* Free integrator memory */
211    CVodeFree(cvode_mem);
212
213    return(0);
214  }
```

```
215
216 /*
217  *-------------------------------
218  * Functions called by the solver
219  *-------------------------------
220  */
221
222
223 /*
224  * f routine. Compute function f(t,y).
225  */
226
227 static void f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
228 {
229   realtype y1, y2, y3, yd1, yd3;
230
231   y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
232
233   yd1 = Ith(ydot,1) = RCONST(-0.04)*y1 + RCONST(1.0e4)*y2*y3;
234   yd3 = Ith(ydot,3) = RCONST(3.0e7)*y2*y2;
235         Ith(ydot,2) = -yd1 - yd3;
236 }
237
238 /*
239  * g routine. Compute functions g_i(t,y) for i = 0,1.
240  */
241
242 static void g(realtype t, N_Vector y, realtype *gout, void *g_data)
243 {
244   realtype y1, y3;
245
246   y1 = Ith(y,1); y3 = Ith(y,3);
247   gout[0] = y1 - RCONST(0.0001);
248   gout[1] = y3 - RCONST(0.01);
249 }
250
251 /*
252  * Jacobian routine. Compute J(t,y) = df/dy. *
253  */
254
255 static void Jac(long int N, DenseMat J, realtype t,
256                 N_Vector y, N_Vector fy, void *jac_data,
257                 N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
258 {
259   realtype y1, y2, y3;
260
261   y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
262
263   IJth(J,1,1) = RCONST(-0.04);
264   IJth(J,1,2) = RCONST(1.0e4)*y3;
265   IJth(J,1,3) = RCONST(1.0e4)*y2;
266   IJth(J,2,1) = RCONST(0.04);
267   IJth(J,2,2) = RCONST(-1.0e4)*y3-RCONST(6.0e7)*y2;
268   IJth(J,2,3) = RCONST(-1.0e4)*y2;
```

```
  IJth(J,3,2) = RCONST(6.0e7)*y2;
}


/*
 *-------------------------------
 * 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, njeD, nfeD, 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, &njeD);
  check_flag(&flag, "CVDenseGetNumJacEvals", 1);
```

```
323   flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeD);
324   check_flag(&flag, "CVDenseGetNumRhsEvals", 1);
325 
326   flag = CVodeGetNumGEvals(cvode_mem, &nge);
327   check_flag(&flag, "CVodeGetNumGEvals", 1);
328 
329   printf("\nFinal Statistics:\n");
330   printf("nst = %-6ld nfe  = %-6ld nsetups = %-6ld nfeD = %-6ld njeD = %ld\n",
331          nst, nfe, nsetups, nfeD, njeD);
332   printf("nni = %-6ld ncfn = %-6ld netf = %-6ld nge = %ld\n \n",
333          nni, ncfn, netf, nge);
334 }
335 
336 /*
337  * Check function return value...
338  *   opt == 0 means SUNDIALS function allocates memory so check if
339  *            returned NULL pointer
340  *   opt == 1 means SUNDIALS function returns a flag so check if
341  *            flag >= 0
342  *   opt == 2 means function allocates memory so check if returned
343  *            NULL pointer
344  */
345 
346 static int check_flag(void *flagvalue, char *funcname, int opt)
347 {
348   int *errflag;
349 
350   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
351   if (opt == 0 && flagvalue == NULL) {
352     fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
353             funcname);
354     return(1); }
355 
356   /* Check if flag < 0 */
357   else if (opt == 1) {
358     errflag = flagvalue;
359     if (*errflag < 0) {
360       fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
361               funcname, *errflag);
362       return(1); }}
363 
364   /* Check if function returned NULL pointer - no memory allocated */
365   else if (opt == 2 && flagvalue == NULL) {
366     fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
367             funcname);
368     return(1); }
369 
370   return(0);
371 }
```

B Listing of cvbx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.17 $
 * $Date: 2004/11/22 23:20:46 $
 * -----------------------------------------------------------------
 * 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 cvbx.c */

#include "sundialstypes.h"   /* definition of type realtype                   */
#include "cvode.h"           /* prototypes for CVode* functions and constants */
                             /* CV_BDF, CV_NEWTON, CV_SS, CV_NORMAL, and      */
                             /* CV_SUCCESS                                    */
#include "cvband.h"          /* prototype for CVBand                          */
#include "nvector_serial.h"  /* definitions of type N_Vector, macro           */
                             /* NV_DATA_S, and prototypes for N_VNew_Serial   */
                             /* and N_VDestroy_Serial                         */
#include "band.h"            /* definitions of type BandMat and macros        */

/* 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    */

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

```
161       CV_SS   specifies scalar relative and absolute tolerances
162       &reltol is a pointer to the scalar relative tolerance
163       &abstol is a pointer to the scalar absolute tolerance vector
164    */
165
166    flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
167    if(check_flag(&flag, "CVodeMalloc", 1)) return(1);
168
169    /* Set the pointer to user-defined data */
170
171    flag = CVodeSetFdata(cvode_mem, data);
172    if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);
173
174    /* Call CVBand to specify the CVBAND band linear solver */
175
176    flag = CVBand(cvode_mem, NEQ, MY, MY);
177    if(check_flag(&flag, "CVBand", 1)) return(1);
178
179    /* Set the user-supplied Jacobian routine Jac and
180       the pointer to the user-defined block data. */
181
182    flag = CVBandSetJacFn(cvode_mem, Jac);
183    if(check_flag(&flag, "CVBandSetJacFn", 1)) return(1);
184    flag = CVBandSetJacData(cvode_mem, data);
185    if(check_flag(&flag, "CVBandSetJacData", 1)) return(1);
186
187    /* In loop over output points: call CVode, print results, test for errors */
188
189    umax = N_VMaxNorm(u);
190    PrintHeader(reltol, abstol, umax);
191    for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
192      flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
193      if(check_flag(&flag, "CVode", 1)) break;
194      umax = N_VMaxNorm(u);
195      flag = CVodeGetNumSteps(cvode_mem, &nst);
196      check_flag(&flag, "CVodeGetNumSteps", 1);
197      PrintOutput(t, umax, nst);
198    }
199
200    PrintFinalStats(cvode_mem);  /* Print some final statistics   */
201
202    N_VDestroy_Serial(u);  /* Free the u vector */
203    CVodeFree(cvode_mem);  /* Free the integrator memory */
204    free(data);            /* Free the user data */
205
206    return(0);
207  }
208
209  /*
210   *-------------------------------
211   * Functions called by the solver
212   *-------------------------------
213   */
214
```

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

static void 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;
    }
  }
}

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

static void 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;

  /*
```

```
269       The components of f = udot that depend on u(i,j) are
270       f(i,j), f(i-1,j), f(i+1,j), f(i,j-1), f(i,j+1), with
271         df(i,j)/du(i,j) = -2 (1/dx^2 + 1/dy^2)
272         df(i-1,j)/du(i,j) = 1/dx^2 + .25/dx  (if i > 1)
273         df(i+1,j)/du(i,j) = 1/dx^2 - .25/dx  (if i < MX)
274         df(i,j-1)/du(i,j) = 1/dy^2           (if j > 1)
275         df(i,j+1)/du(i,j) = 1/dy^2           (if j < MY)
276     */
277
278     data = (UserData) jac_data;
279     hordc = data->hdcoef;
280     horac = data->hacoef;
281     verdc = data->vdcoef;
282
283     for (j=1; j <= MY; j++) {
284       for (i=1; i <= MX; i++) {
285         k = j-1 + (i-1)*MY;
286         kthCol = BAND_COL(J,k);
287
288         /* set the kth column of J */
289
290         BAND_COL_ELEM(kthCol,k,k) = -TWO*(verdc+hordc);
291         if (i != 1)  BAND_COL_ELEM(kthCol,k-MY,k) = hordc + horac;
292         if (i != MX) BAND_COL_ELEM(kthCol,k+MY,k) = hordc - horac;
293         if (j != 1)  BAND_COL_ELEM(kthCol,k-1,k)  = verdc;
294         if (j != MY) BAND_COL_ELEM(kthCol,k+1,k)  = verdc;
295       }
296     }
297   }
298
299   /*
300    *------------------------------
301    * Private helper functions
302    *------------------------------
303    */
304
305   /* Set initial conditions in u vector */
306
307   static void SetIC(N_Vector u, UserData data)
308   {
309     int i, j;
310     realtype x, y, dx, dy;
311     realtype *udata;
312
313     /* Extract needed constants from data */
314
315     dx = data->dx;
316     dy = data->dy;
317
318     /* Set pointer to data array in vector u. */
319
320     udata = NV_DATA_S(u);
321
322     /* Load initial profile into u vector */
```

```
323
324   for (j=1; j <= MY; j++) {
325     y = j*dy;
326     for (i=1; i <= MX; i++) {
327       x = i*dx;
328       IJth(udata,i,j) = x*(XMAX - x)*y*(YMAX - y)*exp(FIVE*x*y);
329     }
330   }
331 }
332
333 /* Print first lines of output (problem description) */
334
335 static void PrintHeader(realtype reltol, realtype abstol, realtype umax)
336 {
337   printf("\n2-D Advection-Diffusion Equation\n");
338   printf("Mesh dimensions = %d X %d\n", MX, MY);
339   printf("Total system size = %d\n", NEQ);
340 #if defined(SUNDIALS_EXTENDED_PRECISION)
341   printf("Tolerance parameters: reltol = %Lg   abstol = %Lg\n\n", reltol, abstol);
342   printf("At t = %Lg      max.norm(u) =%14.6Le \n", T0, umax);
343 #elif defined(SUNDIALS_DOUBLE_PRECISION)
344   printf("Tolerance parameters: reltol = %lg   abstol = %lg\n\n", reltol, abstol);
345   printf("At t = %lg      max.norm(u) =%14.6le \n", T0, umax);
346 #else
347   printf("Tolerance parameters: reltol = %g   abstol = %g\n\n", reltol, abstol);
348   printf("At t = %g      max.norm(u) =%14.6e \n", T0, umax);
349 #endif
350
351   return;
352 }
353
354 /* Print current value */
355
356 static void PrintOutput(realtype t, realtype umax, long int nst)
357 {
358 #if defined(SUNDIALS_EXTENDED_PRECISION)
359   printf("At t = %4.2Lf   max.norm(u) =%14.6Le   nst = %4ld\n", t, umax, nst);
360 #elif defined(SUNDIALS_DOUBLE_PRECISION)
361   printf("At t = %4.2f   max.norm(u) =%14.6le   nst = %4ld\n", t, umax, nst);
362 #else
363   printf("At t = %4.2f   max.norm(u) =%14.6e   nst = %4ld\n", t, umax, nst);
364 #endif
365
366   return;
367 }
368
369 /* Get and print some final statistics */
370
371 static void PrintFinalStats(void *cvode_mem)
372 {
373   int flag;
374   long int nst, nfe, nsetups, netf, nni, ncfn, njeB, nfeB;
375
376   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, &njeB);
  check_flag(&flag, "CVBandGetNumJacEvals", 1);
  flag = CVBandGetNumRhsEvals(cvode_mem, &nfeB);
  check_flag(&flag, "CVBandGetNumRhsEvals", 1);

  printf("\nFinal Statistics:\n");
  printf("nst = %-6ld nfe  = %-6ld nsetups = %-6ld nfeB = %-6ld njeB = %ld\n",
         nst, nfe, nsetups, nfeB, njeB);
  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 = flagvalue;
    if (*errflag < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1); }}

```

```
431     /* Check if function returned NULL pointer - no memory allocated */
432
433     else if (opt == 2 && flagvalue == NULL) {
434       fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
435               funcname);
436       return(1); }
437
438     return(0);
439   }
```

C Listing of cvkx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.17 $
 * $Date: 2004/11/15 18:56:35 $
 * -----------------------------------------------------------------
 * 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 "sundialstypes.h"  /* definitions of realtype, TRUE and FALSE     */
#include "cvode.h"          /* CVode* prototypes and various constants     */
#include "cvspgmr.h"        /* prototypes & constants for CVSPGMR solver   */
#include "smalldense.h"     /* use generic DENSE solver in preconditioning */
#include "nvector_serial.h" /* definitions of type N_Vector and macro      */
                            /* NV_DATA_S                                   */
#include "sundialsmath.h"   /* contains SQR macro                          */

/* 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
```

```
107     work with matrices stored by column in a 2-dimensional array. In C,
108     arrays are indexed starting at 0, not 1. */
109
110  #define IJKth(vdata,i,j,k) (vdata[i-1 + (j)*NUM_SPECIES + (k)*NSMX])
111  #define IJth(a,i,j)        (a[j-1][i-1])
112
113  /* Type : UserData
114     contains preconditioner blocks, pivot arrays, and problem constants */
115
116  typedef struct {
117    realtype **P[MX][MY], **Jbd[MX][MY];
118    long int *pivot[MX][MY];
119    realtype q4, om, dx, dy, hdco, haco, vdco;
120  } *UserData;
121
122  /* Private Helper Functions */
123
124  static UserData AllocUserData(void);
125  static void InitUserData(UserData data);
126  static void FreeUserData(UserData data);
127  static void SetInitialProfiles(N_Vector u, realtype dx, realtype dy);
128  static void PrintOutput(void *cvode_mem, N_Vector u, realtype t);
129  static void PrintFinalStats(void *cvode_mem);
130  static int check_flag(void *flagvalue, char *funcname, int opt);
131
132  /* Functions Called by the Solver */
133
134  static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
135
136  static int Precond(realtype tn, N_Vector u, N_Vector fu,
137                     booleantype jok, booleantype *jcurPtr, realtype gamma,
138                     void *P_data, N_Vector vtemp1, N_Vector vtemp2,
139                     N_Vector vtemp3);
140
141  static int PSolve(realtype tn, N_Vector u, N_Vector fu,
142                    N_Vector r, N_Vector z,
143                    realtype gamma, realtype delta,
144                    int lr, void *P_data, N_Vector vtemp);
145
146
147  /*
148   *--------------------------------
149   * Main Program
150   *--------------------------------
151   */
152
153  int main()
154  {
155    realtype abstol, reltol, t, tout;
156    N_Vector u;
157    UserData data;
158    void *cvode_mem;
159    int iout, flag;
160
```

```
161   u = NULL;
162   data = NULL;
163   cvode_mem = NULL;
164
165   /* Allocate memory, and set problem data, initial values, tolerances */
166   u = N_VNew_Serial(NEQ);
167   if(check_flag((void *)u, "N_VNew_Serial", 0)) return(1);
168   data = AllocUserData();
169   if(check_flag((void *)data, "AllocUserData", 2)) return(1);
170   InitUserData(data);
171   SetInitialProfiles(u, data->dx, data->dy);
172   abstol=ATOL;
173   reltol=RTOL;
174
175   /* Call CvodeCreate to create the solver memory
176
177      CV_BDF     specifies the Backward Differentiation Formula
178      CV_NEWTON  specifies a Newton iteration
179
180      A pointer to the integrator memory is returned and stored in cvode_mem. */
181   cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
182   if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);
183
184   /* Set the pointer to user-defined data */
185   flag = CVodeSetFdata(cvode_mem, data);
186   if(check_flag(&flag, "CVodeSetFdata", 1)) return(1);
187
188   /* Call CVodeMalloc to initialize the integrator memory:
189
190      f       is the user's right hand side function in u'=f(t,u)
191      T0      is the initial time
192      u       is the initial dependent variable vector
193      CV_SS   specifies scalar relative and absolute tolerances
194      &reltol and &abstol are pointers to the scalar tolerances      */
195   flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
196   if(check_flag(&flag, "CVodeMalloc", 1)) return(1);
197
198   /* Call CVSpgmr to specify the linear solver CVSPGMR
199      with left preconditioning and the maximum Krylov dimension maxl */
200   flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
201   if(check_flag(&flag, "CVSpgmr", 1)) return(1);
202
203   /* Set modified Gram-Schmidt orthogonalization, preconditioner
204      setup and solve routines Precond and PSolve, and the pointer
205      to the user-defined block data */
206   flag = CVSpgmrSetGSType(cvode_mem, MODIFIED_GS);
207   if(check_flag(&flag, "CVSpgmrSetGSType", 1)) return(1);
208
209   flag = CVSpgmrSetPrecSetupFn(cvode_mem, Precond);
210   if(check_flag(&flag, "CVSpgmrSetPrecSetupFn", 1)) return(1);
211
212   flag = CVSpgmrSetPrecSolveFn(cvode_mem, PSolve);
213   if(check_flag(&flag, "CVSpgmrSetPrecSolveFn", 1)) return(1);
214
```

```
  flag = CVSpgmrSetPrecData(cvode_mem, data);
  if(check_flag(&flag, "CVSpgmrSetPrecData", 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
 *-------------------------------
 */

/* 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);
      (data->Jbd)[jx][jy] = denalloc(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 lenrwSPGMR, leniwSPGMR;
  long int nst, nfe, nsetups, nni, ncfn, netf;
  long int nli, npe, nps, ncfl, nfeSPGMR;
  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 = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
  check_flag(&flag, "CVSpgmrGetWorkSpace", 1);
  flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
  check_flag(&flag, "CVSpgmrGetNumLinIters", 1);
  flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
  check_flag(&flag, "CVSpgmrGetNumPrecEvals", 1);
  flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
  check_flag(&flag, "CVSpgmrGetNumPrecSolves", 1);
  flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
  check_flag(&flag, "CVSpgmrGetNumConvFails", 1);
  flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);
  check_flag(&flag, "CVSpgmrGetNumRhsEvals", 1);

  printf("\nFinal Statistics.. \n\n");
  printf("lenrw   = %5ld     leniw = %5ld\n", lenrw, leniw);
  printf("llrw    = %5ld     lliw  = %5ld\n", lenrwSPGMR, leniwSPGMR);
  printf("nst     = %5ld\n"                  , nst);
  printf("nfe     = %5ld     nfel  = %5ld\n"  , nfe, nfeSPGMR);
  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 = flagvalue;
```

```
431     if (*errflag < 0) {
432       fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
433               funcname, *errflag);
434       return(1); }}
435 
436   /* Check if function returned NULL pointer - no memory allocated */
437   else if (opt == 2 && flagvalue == NULL) {
438     fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
439             funcname);
440     return(1); }
441 
442   return(0);
443 }
444 
445 /*
446  *-------------------------------
447  * Functions called by the solver
448  *-------------------------------
449  */
450 
451 /* f routine. Compute RHS function f(t,u). */
452 
453 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
454 {
455   realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
456   realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
457   realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
458   realtype q4coef, dely, verdco, hordco, horaco;
459   realtype *udata, *dudata;
460   int jx, jy, idn, iup, ileft, iright;
461   UserData data;
462 
463   data = (UserData) f_data;
464   udata = NV_DATA_S(u);
465   dudata = NV_DATA_S(udot);
466 
467   /* Set diurnal rate coefficients. */
468 
469   s = sin(data->om*t);
470   if (s > ZERO) {
471     q3 = exp(-A3/s);
472     data->q4 = exp(-A4/s);
473   } else {
474       q3 = ZERO;
475       data->q4 = ZERO;
476   }
477 
478   /* Make local copies of problem variables, for efficiency. */
479 
480   q4coef = data->q4;
481   dely = data->dy;
482   verdco = data->vdco;
483   hordco  = data->hdco;
484   horaco  = data->haco;
```

```
485
486     /* Loop over all grid points. */
487
488     for (jy=0; jy < MY; jy++) {
489
490       /* Set vertical diffusion coefficients at jy +- 1/2 */
491
492       ydn = YMIN + (jy - RCONST(0.5))*dely;
493       yup = ydn + dely;
494       cydn = verdco*exp(RCONST(0.2)*ydn);
495       cyup = verdco*exp(RCONST(0.2)*yup);
496       idn = (jy == 0) ? 1 : -1;
497       iup = (jy == MY-1) ? -1 : 1;
498       for (jx=0; jx < MX; jx++) {
499
500         /* Extract c1 and c2, and set kinetic rate terms. */
501
502         c1 = IJKth(udata,1,jx,jy);
503         c2 = IJKth(udata,2,jx,jy);
504         qq1 = Q1*c1*C3;
505         qq2 = Q2*c1*c2;
506         qq3 = q3*C3;
507         qq4 = q4coef*c2;
508         rkin1 = -qq1 - qq2 + TWO*qq3 + qq4;
509         rkin2 = qq1 - qq2 - qq4;
510
511         /* Set vertical diffusion terms. */
512
513         c1dn = IJKth(udata,1,jx,jy+idn);
514         c2dn = IJKth(udata,2,jx,jy+idn);
515         c1up = IJKth(udata,1,jx,jy+iup);
516         c2up = IJKth(udata,2,jx,jy+iup);
517         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
518         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
519
520         /* Set horizontal diffusion and advection terms. */
521
522         ileft = (jx == 0) ? 1 : -1;
523         iright =(jx == MX-1) ? -1 : 1;
524         c1lt = IJKth(udata,1,jx+ileft,jy);
525         c2lt = IJKth(udata,2,jx+ileft,jy);
526         c1rt = IJKth(udata,1,jx+iright,jy);
527         c2rt = IJKth(udata,2,jx+iright,jy);
528         hord1 = hordco*(c1rt - TWO*c1 + c1lt);
529         hord2 = hordco*(c2rt - TWO*c2 + c2lt);
530         horad1 = horaco*(c1rt - c1lt);
531         horad2 = horaco*(c2rt - c2lt);
532
533         /* Load all terms into udot. */
534
535         IJKth(dudata, 1, jx, jy) = vertd1 + hord1 + horad1 + rkin1;
536         IJKth(dudata, 2, jx, jy) = vertd2 + hord2 + horad2 + rkin2;
537       }
538     }
```

```
539
540 }
541
542 /* Preconditioner setup routine. Generate and preprocess P. */
543
544 static int Precond(realtype tn, N_Vector u, N_Vector fu,
545                    booleantype jok, booleantype *jcurPtr, realtype gamma,
546                    void *P_data, N_Vector vtemp1, N_Vector vtemp2,
547                    N_Vector vtemp3)
548 {
549   realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
550   realtype **(*P)[MY], **(*Jbd)[MY];
551   long int *(*pivot)[MY], ier;
552   int jx, jy;
553   realtype *udata, **a, **j;
554   UserData data;
555
556   /* Make local copies of pointers in P_data, and of pointer to u's data */
557
558   data = (UserData) P_data;
559   P = data->P;
560   Jbd = data->Jbd;
561   pivot = data->pivot;
562   udata = NV_DATA_S(u);
563
564   if (jok) {
565
566     /* jok = TRUE: Copy Jbd to P */
567
568     for (jy=0; jy < MY; jy++)
569       for (jx=0; jx < MX; jx++)
570         dencopy(Jbd[jx][jy], P[jx][jy], NUM_SPECIES);
571
572     *jcurPtr = FALSE;
573
574   }
575
576   else {
577     /* jok = FALSE: Generate Jbd from scratch and copy to P */
578
579     /* Make local copies of problem variables, for efficiency. */
580
581     q4coef = data->q4;
582     dely = data->dy;
583     verdco = data->vdco;
584     hordco  = data->hdco;
585
586     /* Compute 2x2 diagonal Jacobian blocks (using q4 values
587        computed on the last f call).  Load into P. */
588
589     for (jy=0; jy < MY; jy++) {
590       ydn = YMIN + (jy - RCONST(0.5))*dely;
591       yup = ydn + dely;
592       cydn = verdco*exp(RCONST(0.2)*ydn);
```

```
593         cyup = verdco*exp(RCONST(0.2)*yup);
594         diag = -(cydn + cyup + TWO*hordco);
595         for (jx=0; jx < MX; jx++) {
596           c1 = IJKth(udata,1,jx,jy);
597           c2 = IJKth(udata,2,jx,jy);
598           j = Jbd[jx][jy];
599           a = P[jx][jy];
600           IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
601           IJth(j,1,2) = -Q2*c1 + q4coef;
602           IJth(j,2,1) = Q1*C3 - Q2*c2;
603           IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
604           dencopy(j, a, NUM_SPECIES);
605         }
606       }
607 
608       *jcurPtr = TRUE;
609 
610     }
611 
612     /* Scale by -gamma */
613 
614     for (jy=0; jy < MY; jy++)
615       for (jx=0; jx < MX; jx++)
616         denscale(-gamma, P[jx][jy], NUM_SPECIES);
617 
618     /* Add identity matrix and do LU decompositions on blocks in place. */
619 
620     for (jx=0; jx < MX; jx++) {
621       for (jy=0; jy < MY; jy++) {
622         denaddI(P[jx][jy], NUM_SPECIES);
623         ier = gefa(P[jx][jy], NUM_SPECIES, pivot[jx][jy]);
624         if (ier != 0) return(1);
625       }
626     }
627 
628     return(0);
629   }
630 
631   /* Preconditioner solve routine */
632 
633   static int PSolve(realtype tn, N_Vector u, N_Vector fu,
634                     N_Vector r, N_Vector z,
635                     realtype gamma, realtype delta,
636                     int lr, void *P_data, N_Vector vtemp)
637   {
638     realtype **(*P)[MY];
639     long int *(*pivot)[MY];
640     int jx, jy;
641     realtype *zdata, *v;
642     UserData data;
643 
644     /* Extract the P and pivot arrays from P_data. */
645 
646     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));
      gesl(P[jx][jy], NUM_SPECIES, pivot[jx][jy], v);
    }
  }

  return(0);
}
```

D Listing of pvnx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.12 $
 * $Date: 2004/11/15 18:56:39 $
 * -----------------------------------------------------------------
 * 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 "sundialstypes.h"     /* definition of realtype                      */
#include "cvode.h"             /* prototypes for CVode* and various constants */
#include "nvector_parallel.h"  /* definitions of type N_Vector and vector     */
                               /* macros, and prototypes for N_Vector         */
                               /* functions                                   */
#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         */
```

```
53 #define DTOUT RCONST(0.5)    /* output time increment     */
54 #define NOUT  10             /* number of output times    */
55
56 /* Type : UserData
57    contains grid constants, parallel machine parameters, work array. */
58
59 typedef struct {
60   realtype dx, hdcoef, hacoef;
61   int npes, my_pe;
62   MPI_Comm comm;
63   realtype z[100];
64 } *UserData;
65
66 /* Private Helper Functions */
67
68 static void SetIC(N_Vector u, realtype dx, long int my_length,
69                   long int my_base);
70
71 static void PrintIntro(int npes);
72
73 static void PrintData(realtype t, realtype umax, long int nst);
74
75 static void PrintFinalStats(void *cvode_mem);
76
77 /* Functions Called by the Solver */
78
79 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
80
81 /* Private function to check function return values */
82
83 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
84
85 /***************************** Main Program ******************************/
86
87 int main(int argc, char *argv[])
88 {
89   realtype dx, reltol, abstol, t, tout, umax;
90   N_Vector u;
91   UserData data;
92   void *cvode_mem;
93   int iout, flag, my_pe, npes;
94   long int local_N, nperpe, nrem, my_base, nst;
95
96   MPI_Comm comm;
97
98   u = NULL;
99   data = NULL;
100  cvode_mem = NULL;
101
102  /* Get processor number, total number of pe's, and my_pe. */
103  MPI_Init(&argc, &argv);
104  comm = MPI_COMM_WORLD;
105  MPI_Comm_size(comm, &npes);
106  MPI_Comm_rank(comm, &my_pe);
```

```
107
108   /* Set local vector length. */
109   nperpe = NEQ/npes;
110   nrem = NEQ - npes*nperpe;
111   local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
112   my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;
113
114   data = (UserData) malloc(sizeof *data);  /* Allocate data memory */
115   if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
116
117   data->comm = comm;
118   data->npes = npes;
119   data->my_pe = my_pe;
120
121   u = N_VNew_Parallel(comm, local_N, NEQ);  /* Allocate u vector */
122   if(check_flag((void *)u, "N_VNew", 0, my_pe)) MPI_Abort(comm, 1);
123
124   reltol = ZERO;  /* Set the tolerances */
125   abstol = ATOL;
126
127   dx = data->dx = XMAX/((realtype)(MX+1));  /* Set grid coefficients in data */
128   data->hdcoef = RCONST(1.0)/(dx*dx);
129   data->hacoef = RCONST(0.5)/(RCONST(2.0)*dx);
130
131   SetIC(u, dx, local_N, my_base);  /* Initialize u vector */
132
133   /*
134      Call CVodeCreate to create the solver memory:
135
136      CV_ADAMS   specifies the Adams Method
137      CV_FUNCTIONAL  specifies functional iteration
138
139      A pointer to the integrator memory is returned and stored in cvode_mem.
140   */
141
142   cvode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
143   if(check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
144
145   flag = CVodeSetFdata(cvode_mem, data);
146   if(check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
147
148   /*
149      Call CVodeMalloc to initialize the integrator memory:
150
151      cvode_mem is the pointer to the integrator memory returned by CVodeCreate
152      f       is the user's right hand side function in y'=f(t,y)
153      T0      is the initial time
154      u       is the initial dependent variable vector
155      CV_SS   specifies scalar relative and absolute tolerances
156      &reltol and &abstol are pointers to the scalar tolerances
157   */
158
159   flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
160   if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
```

```
161
162   if (my_pe == 0) PrintIntro(npes);
163
164   umax = N_VMaxNorm(u);
165
166   if (my_pe == 0) PrintData(t, umax, 0);
167
168   /* In loop over output points, call CVode, print results, test for error */
169
170   for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
171     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
172     if(check_flag(&flag, "CVode", 1, my_pe)) break;
173     umax = N_VMaxNorm(u);
174     flag = CVodeGetNumSteps(cvode_mem, &nst);
175     check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
176     if (my_pe == 0) PrintData(t, umax, nst);
177   }
178
179   if (my_pe == 0)
180     PrintFinalStats(cvode_mem);  /* Print some final statistics */
181
182   N_VDestroy_Parallel(u);        /* Free the u vector */
183   CVodeFree(cvode_mem);          /* Free the integrator memory */
184   free(data);                    /* Free user data */
185
186   MPI_Finalize();
187
188   return(0);
189 }
190
191 /*********************** Private Helper Functions ***********************/
192
193 /* Set initial conditions in u vector */
194
195 static void SetIC(N_Vector u, realtype dx, long int my_length,
196                   long int my_base)
197 {
198   int i;
199   long int iglobal;
200   realtype x;
201   realtype *udata;
202
203   /* Set pointer to data array and get local length of u. */
204   udata = NV_DATA_P(u);
205   my_length = NV_LOCLENGTH_P(u);
206
207   /* Load initial profile into u vector */
208   for (i=1; i<=my_length; i++) {
209     iglobal = my_base + i;
210     x = iglobal*dx;
211     udata[i-1] = x*(XMAX - x)*exp(RCONST(2.0)*x);
212   }
213 }
214
```

```
215 /* Print problem introduction */
216
217 static void PrintIntro(int npes)
218 {
219   printf("\n 1-D advection-diffusion equation, mesh size =%3d \n", MX);
220   printf("\n Number of PEs = %3d \n\n", npes);
221
222   return;
223 }
224
225 /* Print data */
226
227 static void PrintData(realtype t, realtype umax, long int nst)
228 {
229
230 #if defined(SUNDIALS_EXTENDED_PRECISION)
231   printf("At t = %4.2Lf  max.norm(u) =%14.6Le  nst =%4ld \n", t, umax, nst);
232 #elif defined(SUNDIALS_DOUBLE_PRECISION)
233   printf("At t = %4.2f  max.norm(u) =%14.6le  nst =%4ld \n", t, umax, nst);
234 #else
235   printf("At t = %4.2f  max.norm(u) =%14.6e  nst =%4ld \n", t, umax, nst);
236 #endif
237
238   return;
239 }
240
241 /* Print some final statistics located in the iopt array */
242
243 static void PrintFinalStats(void *cvode_mem)
244 {
245   long int nst, nfe, nni, ncfn, netf;
246   int flag;
247
248   flag = CVodeGetNumSteps(cvode_mem, &nst);
249   check_flag(&flag, "CVodeGetNumSteps", 1, 0);
250   flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
251   check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
252   flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
253   check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
254   flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
255   check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
256   flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
257   check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
258
259   printf("\nFinal Statistics: \n\n");
260   printf("nst = %-6ld  nfe  = %-6ld  ", nst, nfe);
261   printf("nni = %-6ld  ncfn = %-6ld  netf = %ld\n \n", nni, ncfn, netf);
262 }
263
264 /****************** Function Called by the Solver ***********************/
265
266 /* f routine. Compute f(t,u). */
267
268 static void 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];
```

```
323       ult = z[i-1];
324       urt = z[i+1];
325
326       /* Set diffusion and advection terms and load into udot */
327       hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
328       hadv = horac*(urt - ult);
329       dudata[i-1] = hdiff + hadv;
330     }
331   }
332
333   /* Check function return value...
334        opt == 0 means SUNDIALS function allocates memory so check if
335                 returned NULL pointer
336        opt == 1 means SUNDIALS function returns a flag so check if
337                 flag >= 0
338        opt == 2 means function allocates memory so check if returned
339                 NULL pointer */
340
341   static int check_flag(void *flagvalue, char *funcname, int opt, int id)
342   {
343     int *errflag;
344
345     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
346     if (opt == 0 && flagvalue == NULL) {
347       fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
348               id, funcname);
349       return(1); }
350
351     /* Check if flag < 0 */
352     else if (opt == 1) {
353       errflag = flagvalue;
354       if (*errflag < 0) {
355         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
356                 id, funcname, *errflag);
357         return(1); }}
358
359     /* Check if function returned NULL pointer - no memory allocated */
360     else if (opt == 2 && flagvalue == NULL) {
361       fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
362               id, funcname);
363       return(1); }
364
365     return(0);
366   }
```

E Listing of pvkx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.14 $
 * $Date: 2004/11/15 18:56:39 $
 * -----------------------------------------------------------------
 * 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 pvkx   with N = NPEX*NPEY (see constants
 * below).
 * -----------------------------------------------------------------
 */

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

```
53 #include <math.h>
54 #include "sundialstypes.h"     /* definitions of realtype, booleantype, TRUE, */
55                                /* and FALSE                                   */
56 #include "sundialsmath.h"      /* definition of macro SQR                     */
57 #include "cvode.h"             /* prototypes for CVode* and various constants */
58 #include "cvspgmr.h"           /* prototypes and constants for CVSPGMR solver */
59 #include "smalldense.h"        /* prototypes for small dense matrix functions */
60 #include "nvector_parallel.h"  /* definition of type N_Vector and macro       */
61                                /* NV_DATA_P                                   */
62 #include "mpi.h"               /* MPI constants and types                     */
63
64 /* Problem Constants */
65
66 #define NVARS        2                   /* number of species         */
67 #define KH           RCONST(4.0e-6)      /* horizontal diffusivity Kh */
68 #define VEL          RCONST(0.001)       /* advection velocity V      */
69 #define KV0          RCONST(1.0e-8)      /* coefficient in Kv(y)      */
70 #define Q1           RCONST(1.63e-16)    /* coefficients q1, q2, c3   */
71 #define Q2           RCONST(4.66e-16)
72 #define C3           RCONST(3.7e16)
73 #define A3           RCONST(22.62)     /* coefficient in expression for q3(t) */
74 #define A4           RCONST(7.601)     /* coefficient in expression for q4(t) */
75 #define C1_SCALE     RCONST(1.0e6)     /* coefficients in initial profiles    */
76 #define C2_SCALE     RCONST(1.0e12)
77
78 #define T0           RCONST(0.0)         /* initial time */
79 #define NOUT         12                  /* number of output times */
80 #define TWOHR        RCONST(7200.0)      /* number of seconds in two hours  */
81 #define HALFDAY      RCONST(4.32e4)      /* number of seconds in a half day */
82 #define PI       RCONST(3.1415926535898)  /* pi */
83
84 #define XMIN         RCONST(0.0)         /* grid boundaries in x  */
85 #define XMAX         RCONST(20.0)
86 #define YMIN         RCONST(30.0)        /* grid boundaries in y  */
87 #define YMAX         RCONST(50.0)
88
89 #define NPEX         2             /* no. PEs in x direction of PE array */
90 #define NPEY         2             /* no. PEs in y direction of PE array */
91                                    /* Total no. PEs = NPEX*NPEY */
92 #define MXSUB        5             /* no. x points per subgrid */
93 #define MYSUB        5             /* no. y points per subgrid */
94
95 #define MX           (NPEX*MXSUB)  /* MX = number of x mesh points */
96 #define MY           (NPEY*MYSUB)  /* MY = number of y mesh points */
97                                    /* Spatial mesh is MX by MY */
98 /* CVodeMalloc Constants */
99
100 #define RTOL    RCONST(1.0e-5)    /* scalar relative tolerance */
101 #define FLOOR   RCONST(100.0)     /* value of C1 or C2 at which tolerances */
102                                   /* change from relative to absolute      */
103 #define ATOL    (RTOL*FLOOR)      /* scalar absolute tolerance */
104
105
106 /* User-defined matrix accessor macro: IJth */
```

```
107
108 /* IJth is defined in order to write code which indexes into small dense
109    matrices with a (row,column) pair, where 1 <= row,column <= NVARS.
110
111    IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
112    where 1 <= i,j <= NVARS. The small matrix routines in dense.h
113    work with matrices stored by column in a 2-dimensional array. In C,
114    arrays are indexed starting at 0, not 1. */
115
116 #define IJth(a,i,j) (a[j-1][i-1])
117
118 /* Type : UserData
119    contains problem constants, preconditioner blocks, pivot arrays,
120    grid constants, and processor indices */
121
122 typedef struct {
123   realtype q4, om, dx, dy, hdco, haco, vdco;
124   realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
125   int my_pe, isubx, isuby;
126   long int nvmxsub, nvmxsub2;
127   MPI_Comm comm;
128 } *UserData;
129
130 typedef struct {
131   void *f_data;
132   realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
133   long int *pivot[MXSUB][MYSUB];
134 } *PreconData;
135
136
137 /* Private Helper Functions */
138
139 static PreconData AllocPreconData(UserData data);
140 static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
141 static void FreePreconData(PreconData pdata);
142 static void SetInitialProfiles(N_Vector u, UserData data);
143 static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
144                         N_Vector u, realtype t);
145 static void PrintFinalStats(void *cvode_mem);
146 static void BSend(MPI_Comm comm,
147                   int my_pe, int isubx, int isuby,
148                   long int dsizex, long int dsizey,
149                   realtype udata[]);
150 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
151                       int my_pe, int isubx, int isuby,
152                       long int dsizex, long int dsizey,
153                       realtype uext[], realtype buffer[]);
154 static void BRecvWait(MPI_Request request[],
155                       int isubx, int isuby,
156                       long int dsizex, realtype uext[],
157                       realtype buffer[]);
158 static void ucomm(realtype t, N_Vector u, UserData data);
159 static void fcalc(realtype t, realtype udata[], realtype dudata[],
160                   UserData data);
```

```
161
162
163 /* Functions Called by the Solver */
164
165 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
166
167 static int Precond(realtype tn, N_Vector u, N_Vector fu,
168                    booleantype jok, booleantype *jcurPtr,
169                    realtype gamma, void *P_data,
170                    N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
171
172 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
173                   N_Vector r, N_Vector z,
174                   realtype gamma, realtype delta,
175                   int lr, void *P_data, N_Vector vtemp);
176
177
178 /* Private function to check function return values */
179
180 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
181
182
183 /***************************** Main Program ****************************/
184
185 int main(int argc, char *argv[])
186 {
187   realtype abstol, reltol, t, tout;
188   N_Vector u;
189   UserData data;
190   PreconData predata;
191   void *cvode_mem;
192   int iout, flag, my_pe, npes;
193   long int neq, local_N;
194   MPI_Comm comm;
195
196   u = NULL;
197   data = NULL;
198   predata = NULL;
199   cvode_mem = NULL;
200
201   /* Set problem size neq */
202   neq = NVARS*MX*MY;
203
204   /* Get processor number and total number of pe's */
205   MPI_Init(&argc, &argv);
206   comm = MPI_COMM_WORLD;
207   MPI_Comm_size(comm, &npes);
208   MPI_Comm_rank(comm, &my_pe);
209
210   if (npes != NPEX*NPEY) {
211     if (my_pe == 0)
212       fprintf(stderr, "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n\n",
213               npes,NPEX*NPEY);
214     MPI_Finalize();
```

```
215       return(1);
216     }
217
218     /* Set local length */
219     local_N = NVARS*MXSUB*MYSUB;
220
221     /* Allocate and load user data block; allocate preconditioner block */
222     data = (UserData) malloc(sizeof *data);
223     if (check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
224     InitUserData(my_pe, comm, data);
225     predata = AllocPreconData (data);
226
227     /* Allocate u, and set initial values and tolerances */
228     u = N_VNew_Parallel(comm, local_N, neq);
229     if (check_flag((void *)u, "N_VNew", 0, my_pe)) MPI_Abort(comm, 1);
230     SetInitialProfiles(u, data);
231     abstol = ATOL; reltol = RTOL;
232
233     /*
234        Call CVodeCreate to create the solver memory:
235
236        CV_BDF     specifies the Backward Differentiation Formula
237        CV_NEWTON  specifies a Newton iteration
238
239        A pointer to the integrator memory is returned and stored in cvode_mem.
240     */
241     cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
242     if (check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
243
244     /* Set the pointer to user-defined data */
245     flag = CVodeSetFdata(cvode_mem, data);
246     if (check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
247
248     /*
249        Call CVodeMalloc to initialize the integrator memory:
250
251        cvode_mem is the pointer to the integrator memory returned by CVodeCreate
252        f       is the user's right hand side function in y'=f(t,y)
253        T0      is the initial time
254        u       is the initial dependent variable vector
255        CV_SS   specifies scalar relative and absolute tolerances
256        &reltol and &abstol are pointers to the scalar tolerances
257     */
258     flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
259     if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
260
261     /* Call CVSpgmr to specify the linear solver CVSPGMR
262        with left preconditioning and the maximum Krylov dimension maxl */
263     flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
264     if (check_flag(&flag, "CVSpgmr", 1, my_pe)) MPI_Abort(comm, 1);
265
266     /* Set preconditioner setup and solve routines Precond and PSolve,
267        and the pointer to the user-defined block data */
268     flag = CVSpgmrSetPrecSetupFn(cvode_mem, Precond);
```

```
269   if (check_flag(&flag, "CVSpgmrSetPrecSetupFn", 1, my_pe)) MPI_Abort(comm, 1);
270
271   flag = CVSpgmrSetPrecSolveFn(cvode_mem, PSolve);
272   if (check_flag(&flag, "CVSpgmrSetPrecSolveFn", 1, my_pe)) MPI_Abort(comm, 1);
273
274   flag = CVSpgmrSetPrecData(cvode_mem, predata);
275   if (check_flag(&flag, "CVSpgmrSetPrecData", 1, my_pe)) MPI_Abort(comm, 1);
276
277   if (my_pe == 0)
278     printf("\n2-species diurnal advection-diffusion problem\n\n");
279
280   /* In loop over output points, call CVode, print results, test for error */
281   for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
282     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
283     if (check_flag(&flag, "CVode", 1, my_pe)) break;
284     PrintOutput(cvode_mem, my_pe, comm, u, t);
285   }
286
287   /* Print final statistics */
288   if (my_pe == 0) PrintFinalStats(cvode_mem);
289
290   /* Free memory */
291   N_VDestroy_Parallel(u);
292   free(data);
293   FreePreconData(predata);
294   CVodeFree(cvode_mem);
295
296   MPI_Finalize();
297
298   return(0);
299 }
300
301
302 /********************** Private Helper Functions **********************/
303
304 /* Allocate memory for data structure of type UserData */
305
306 static PreconData AllocPreconData(UserData fdata)
307 {
308   int lx, ly;
309   PreconData pdata;
310
311   pdata = (PreconData) malloc(sizeof *pdata);
312
313   pdata->f_data = fdata;
314
315   for (lx = 0; lx < MXSUB; lx++) {
316     for (ly = 0; ly < MYSUB; ly++) {
317       (pdata->P)[lx][ly] = denalloc(NVARS);
318       (pdata->Jbd)[lx][ly] = denalloc(NVARS);
319       (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
320     }
321   }
322
```

```
  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);
```

```
431       else {
432         tempu[0] = udata[i0];
433         tempu[1] = udata[i1];
434       }
435     }
436
437     /* On PE 0, receive c1,c2 at top right, then print performance data
438        and sampled solution values */
439     if (my_pe == 0) {
440       if (npelast != 0)
441         MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
442       flag = CVodeGetNumSteps(cvode_mem, &nst);
443       check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
444       flag = CVodeGetLastOrder(cvode_mem, &qu);
445       check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
446       flag = CVodeGetLastStep(cvode_mem, &hu);
447       check_flag(&flag, "CVodeGetLastStep", 1, my_pe);
448
449 #if defined(SUNDIALS_EXTENDED_PRECISION)
450       printf("t = %.2Le   no. steps = %ld   order = %d   stepsize = %.2Le\n",
451              t, nst, qu, hu);
452       printf("At bottom left:  c1, c2 = %12.3Le %12.3Le \n", udata[0], udata[1]);
453       printf("At top right:    c1, c2 = %12.3Le %12.3Le \n\n", tempu[0], tempu[1]);
454 #elif defined(SUNDIALS_DOUBLE_PRECISION)
455       printf("t = %.2le   no. steps = %ld   order = %d   stepsize = %.2le\n",
456              t, nst, qu, hu);
457       printf("At bottom left:  c1, c2 = %12.3le %12.3le \n", udata[0], udata[1]);
458       printf("At top right:    c1, c2 = %12.3le %12.3le \n\n", tempu[0], tempu[1]);
459 #else
460       printf("t = %.2e   no. steps = %ld   order = %d   stepsize = %.2e\n",
461              t, nst, qu, hu);
462       printf("At bottom left:  c1, c2 = %12.3e %12.3e \n", udata[0], udata[1]);
463       printf("At top right:    c1, c2 = %12.3e %12.3e \n\n", tempu[0], tempu[1]);
464 #endif
465     }
466 }
467
468 /* Print final statistics contained in iopt */
469
470 static void PrintFinalStats(void *cvode_mem)
471 {
472   long int lenrw, leniw ;
473   long int lenrwSPGMR, leniwSPGMR;
474   long int nst, nfe, nsetups, nni, ncfn, netf;
475   long int nli, npe, nps, ncfl, nfeSPGMR;
476   int flag;
477
478   flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
479   check_flag(&flag, "CVodeGetWorkSpace", 1, 0);
480   flag = CVodeGetNumSteps(cvode_mem, &nst);
481   check_flag(&flag, "CVodeGetNumSteps", 1, 0);
482   flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
483   check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
484   flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
```

```
485   check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
486   flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
487   check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
488   flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
489   check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
490   flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
491   check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
492
493   flag = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
494   check_flag(&flag, "CVSpgmrGetWorkSpace", 1, 0);
495   flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
496   check_flag(&flag, "CVSpgmrGetNumLinIters", 1, 0);
497   flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
498   check_flag(&flag, "CVSpgmrGetNumPrecEvals", 1, 0);
499   flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
500   check_flag(&flag, "CVSpgmrGetNumPrecSolves", 1, 0);
501   flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
502   check_flag(&flag, "CVSpgmrGetNumConvFails", 1, 0);
503   flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);
504   check_flag(&flag, "CVSpgmrGetNumRhsEvals", 1, 0);
505
506   printf("\nFinal Statistics: \n\n");
507   printf("lenrw   = %5ld     leniw = %5ld\n", lenrw, leniw);
508   printf("llrw    = %5ld     lliw  = %5ld\n", lenrwSPGMR, leniwSPGMR);
509   printf("nst     = %5ld\n"                 , nst);
510   printf("nfe     = %5ld     nfel  = %5ld\n"  , nfe, nfeSPGMR);
511   printf("nni     = %5ld     nli   = %5ld\n"  , nni, nli);
512   printf("nsetups = %5ld     netf  = %5ld\n"  , nsetups, netf);
513   printf("npe     = %5ld     nps   = %5ld\n"  , npe, nps);
514   printf("ncfn    = %5ld     ncfl  = %5ld\n\n", ncfn, ncfl);
515 }
516
517 /* Routine to send boundary data to neighboring PEs */
518
519 static void BSend(MPI_Comm comm,
520                   int my_pe, int isubx, int isuby,
521                   long int dsizex, long int dsizey,
522                   realtype udata[])
523 {
524   int i, ly;
525   long int offsetu, offsetbuf;
526   realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
527
528   /* If isuby > 0, send data from bottom x-line of u */
529   if (isuby != 0)
530     MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
531
532   /* If isuby < NPEY-1, send data from top x-line of u */
533   if (isuby != NPEY-1) {
534     offsetu = (MYSUB-1)*dsizex;
535     MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
536   }
537
538   /* If isubx > 0, send data from left y-line of u (via bufleft) */
```

```
539   if (isubx != 0) {
540     for (ly = 0; ly < MYSUB; ly++) {
541       offsetbuf = ly*NVARS;
542       offsetu = ly*dsizex;
543       for (i = 0; i < NVARS; i++)
544         bufleft[offsetbuf+i] = udata[offsetu+i];
545     }
546     MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
547   }
548 
549   /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
550   if (isubx != NPEX-1) {
551     for (ly = 0; ly < MYSUB; ly++) {
552       offsetbuf = ly*NVARS;
553       offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
554       for (i = 0; i < NVARS; i++)
555         bufright[offsetbuf+i] = udata[offsetu+i];
556     }
557     MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
558   }
559 }
560 
561 /* Routine to start receiving boundary data from neighboring PEs.
562    Notes:
563    1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
564    passed to both the BRecvPost and BRecvWait functions, and should not
565    be manipulated between the two calls.
566    2) request should have 4 entries, and should be passed in both calls also. */
567 
568 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
569                       int my_pe, int isubx, int isuby,
570                       long int dsizex, long int dsizey,
571                       realtype uext[], realtype buffer[])
572 {
573   long int offsetue;
574   /* Have bufleft and bufright use the same buffer */
575   realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
576 
577   /* If isuby > 0, receive data for bottom x-line of uext */
578   if (isuby != 0)
579     MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
580                                           my_pe-NPEX, 0, comm, &request[0]);
581 
582   /* If isuby < NPEY-1, receive data for top x-line of uext */
583   if (isuby != NPEY-1) {
584     offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
585     MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
586                                         my_pe+NPEX, 0, comm, &request[1]);
587   }
588 
589   /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
590   if (isubx != 0) {
591     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
592                                         my_pe-1, 0, comm, &request[2]);
```

```
593     }
594
595     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
596     if (isubx != NPEX-1) {
597       MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
598                                           my_pe+1, 0, comm, &request[3]);
599     }
600   }
601
602   /* Routine to finish receiving boundary data from neighboring PEs.
603      Notes:
604      1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
605      passed to both the BRecvPost and BRecvWait functions, and should not
606      be manipulated between the two calls.
607      2) request should have 4 entries, and should be passed in both calls also. */
608
609   static void BRecvWait(MPI_Request request[],
610                         int isubx, int isuby,
611                         long int dsizex, realtype uext[],
612                         realtype buffer[])
613   {
614     int i, ly;
615     long int dsizex2, offsetue, offsetbuf;
616     realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
617     MPI_Status status;
618
619     dsizex2 = dsizex + 2*NVARS;
620
621     /* If isuby > 0, receive data for bottom x-line of uext */
622     if (isuby != 0)
623       MPI_Wait(&request[0],&status);
624
625     /* If isuby < NPEY-1, receive data for top x-line of uext */
626     if (isuby != NPEY-1)
627       MPI_Wait(&request[1],&status);
628
629     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
630     if (isubx != 0) {
631       MPI_Wait(&request[2],&status);
632
633       /* Copy the buffer to uext */
634       for (ly = 0; ly < MYSUB; ly++) {
635         offsetbuf = ly*NVARS;
636         offsetue = (ly+1)*dsizex2;
637         for (i = 0; i < NVARS; i++)
638           uext[offsetue+i] = bufleft[offsetbuf+i];
639       }
640     }
641
642     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
643     if (isubx != NPEX-1) {
644       MPI_Wait(&request[3],&status);
645
646       /* Copy the buffer to uext */
```

```
647       for (ly = 0; ly < MYSUB; ly++) {
648         offsetbuf = ly*NVARS;
649         offsetue = (ly+2)*dsizex2 - NVARS;
650         for (i = 0; i < NVARS; i++)
651           uext[offsetue+i] = bufright[offsetbuf+i];
652       }
653     }
654   }
655
656   /* ucomm routine.  This routine performs all communication
657      between processors of data needed to calculate f. */
658
659   static void ucomm(realtype t, N_Vector u, UserData data)
660   {
661
662     realtype *udata, *uext, buffer[2*NVARS*MYSUB];
663     MPI_Comm comm;
664     int my_pe, isubx, isuby;
665     long int nvmxsub, nvmysub;
666     MPI_Request request[4];
667
668     udata = NV_DATA_P(u);
669
670     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
671     comm = data->comm;  my_pe = data->my_pe;
672     isubx = data->isubx;   isuby = data->isuby;
673     nvmxsub = data->nvmxsub;
674     nvmysub = NVARS*MYSUB;
675     uext = data->uext;
676
677     /* Start receiving boundary data from neighboring PEs */
678     BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);
679
680     /* Send data from boundary of local grid to neighboring PEs */
681     BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);
682
683     /* Finish receiving boundary data from neighboring PEs */
684     BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
685   }
686
687   /* fcalc routine. Compute f(t,y).  This routine assumes that communication
688      between processors of data needed to calculate f has already been done,
689      and this data is in the work array uext. */
690
691   static void fcalc(realtype t, realtype udata[],
692                     realtype dudata[], UserData data)
693   {
694     realtype *uext;
695     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
696     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
697     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
698     realtype q4coef, dely, verdco, hordco, horaco;
699     int i, lx, ly, jx, jy;
700     int isubx, isuby;
```

```
701   long int nvmxsub, nvmxsub2, offsetu, offsetue;
702
703   /* Get subgrid indices, data sizes, extended work array uext */
704   isubx = data->isubx;   isuby = data->isuby;
705   nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
706   uext = data->uext;
707
708   /* Copy local segment of u vector into the working extended array uext */
709   offsetu = 0;
710   offsetue = nvmxsub2 + NVARS;
711   for (ly = 0; ly < MYSUB; ly++) {
712     for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
713     offsetu = offsetu + nvmxsub;
714     offsetue = offsetue + nvmxsub2;
715   }
716
717   /* To facilitate homogeneous Neumann boundary conditions, when this is
718   a boundary PE, copy data from the first interior mesh line of u to uext */
719
720   /* If isuby = 0, copy x-line 2 of u to uext */
721   if (isuby == 0) {
722     for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = udata[nvmxsub+i];
723   }
724
725   /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
726   if (isuby == NPEY-1) {
727     offsetu = (MYSUB-2)*nvmxsub;
728     offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
729     for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
730   }
731
732   /* If isubx = 0, copy y-line 2 of u to uext */
733   if (isubx == 0) {
734     for (ly = 0; ly < MYSUB; ly++) {
735       offsetu = ly*nvmxsub + NVARS;
736       offsetue = (ly+1)*nvmxsub2;
737       for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
738     }
739   }
740
741   /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
742   if (isubx == NPEX-1) {
743     for (ly = 0; ly < MYSUB; ly++) {
744       offsetu = (ly+1)*nvmxsub - 2*NVARS;
745       offsetue = (ly+2)*nvmxsub2 - NVARS;
746       for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
747     }
748   }
749
750   /* Make local copies of problem variables, for efficiency */
751   dely = data->dy;
752   verdco = data->vdco;
753   hordco  = data->hdco;
754   horaco  = data->haco;
```

```
755
756     /* Set diurnal rate coefficients as functions of t, and save q4 in
757     data block for use by preconditioner evaluation routine */
758     s = sin((data->om)*t);
759     if (s > RCONST(0.0)) {
760       q3 = exp(-A3/s);
761       q4coef = exp(-A4/s);
762     } else {
763       q3 = RCONST(0.0);
764       q4coef = RCONST(0.0);
765     }
766     data->q4 = q4coef;
767
768     /* Loop over all grid points in local subgrid */
769     for (ly = 0; ly < MYSUB; ly++) {
770
771       jy = ly + isuby*MYSUB;
772
773       /* Set vertical diffusion coefficients at jy +- 1/2 */
774       ydn = YMIN + (jy - RCONST(0.5))*dely;
775       yup = ydn + dely;
776       cydn = verdco*exp(RCONST(0.2)*ydn);
777       cyup = verdco*exp(RCONST(0.2)*yup);
778       for (lx = 0; lx < MXSUB; lx++) {
779
780         jx = lx + isubx*MXSUB;
781
782         /* Extract c1 and c2, and set kinetic rate terms */
783         offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
784         c1 = uext[offsetue];
785         c2 = uext[offsetue+1];
786         qq1 = Q1*c1*C3;
787         qq2 = Q2*c1*c2;
788         qq3 = q3*C3;
789         qq4 = q4coef*c2;
790         rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
791         rkin2 = qq1 - qq2 - qq4;
792
793         /* Set vertical diffusion terms */
794         c1dn = uext[offsetue-nvmxsub2];
795         c2dn = uext[offsetue-nvmxsub2+1];
796         c1up = uext[offsetue+nvmxsub2];
797         c2up = uext[offsetue+nvmxsub2+1];
798         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
799         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
800
801         /* Set horizontal diffusion and advection terms */
802         c1lt = uext[offsetue-2];
803         c2lt = uext[offsetue-1];
804         c1rt = uext[offsetue+2];
805         c2rt = uext[offsetue+3];
806         hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
807         hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
808         horad1 = horaco*(c1rt - c1lt);
```

```
809         horad2 = horaco*(c2rt - c2lt);
810
811         /* Load all terms into dudata */
812         offsetu = lx*NVARS + ly*nvmxsub;
813         dudata[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
814         dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
815       }
816     }
817   }
818
819
820   /***************** Functions Called by the Solver ************************/
821
822   /* f routine.  Evaluate f(t,y).  First call ucomm to do communication of
823      subgrid boundary data into uext.  Then calculate f by a call to fcalc. */
824
825   static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
826   {
827     realtype *udata, *dudata;
828     UserData data;
829
830     udata = NV_DATA_P(u);
831     dudata = NV_DATA_P(udot);
832     data = (UserData) f_data;
833
834     /* Call ucomm to do inter-processor communication */
835     ucomm (t, u, data);
836
837     /* Call fcalc to calculate all right-hand sides */
838     fcalc (t, udata, dudata, data);
839   }
840
841   /* Preconditioner setup routine. Generate and preprocess P. */
842   static int Precond(realtype tn, N_Vector u, N_Vector fu,
843                      booleantype jok, booleantype *jcurPtr,
844                      realtype gamma, void *P_data,
845                      N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
846   {
847     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
848     realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
849     long int nvmxsub, *(*pivot)[MYSUB], ier, offset;
850     int lx, ly, jx, jy, isubx, isuby;
851     realtype *udata, **a, **j;
852     PreconData predata;
853     UserData data;
854
855     /* Make local copies of pointers in P_data, pointer to u's data,
856        and PE index pair */
857     predata = (PreconData) P_data;
858     data = (UserData) (predata->f_data);
859     P = predata->P;
860     Jbd = predata->Jbd;
861     pivot = predata->pivot;
862     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);

  *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);
      }
    }

  *jcurPtr = TRUE;

  }

  /* Scale by -gamma */
    for (ly = 0; ly < MYSUB; ly++)
```

```
      for (lx = 0; lx < MXSUB; lx++)
        denscale(-gamma, P[lx][ly], 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 = gefa(P[lx][ly], 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]);
      gesl(P[lx][ly], NVARS, pivot[lx][ly], v);
    }
  }

  return(0);
}


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

```
971
972  /* Check function return value...
973       opt == 0 means SUNDIALS function allocates memory so check if
974                returned NULL pointer
975       opt == 1 means SUNDIALS function returns a flag so check if
976                flag >= 0
977       opt == 2 means function allocates memory so check if returned
978                NULL pointer */
979
980  static int check_flag(void *flagvalue, char *funcname, int opt, int id)
981  {
982    int *errflag;
983
984    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
985    if (opt == 0 && flagvalue == NULL) {
986      fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
987              id, funcname);
988      return(1); }
989
990    /* Check if flag < 0 */
991    else if (opt == 1) {
992      errflag = flagvalue;
993      if (*errflag < 0) {
994        fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
995                id, funcname, *errflag);
996        return(1); }}
997
998    /* Check if function returned NULL pointer - no memory allocated */
999    else if (opt == 2 && flagvalue == NULL) {
1000     fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
1001             id, funcname);
1002     return(1); }
1003
1004   return(0);
1005 }
```

F Listing of pvkxb.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.19 $
 * $Date: 2004/11/15 18:56:39 $
 * -----------------------------------------------------------------
 * 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).
 * -----------------------------------------------------------------
 */
```

```
53
54 #include <stdio.h>
55 #include <stdlib.h>
56 #include <math.h>
57 #include "sundialstypes.h"     /* definition of type realtype                */
58 #include "sundialsmath.h"      /* definition of macro SQR                    */
59 #include "cvode.h"             /* prototypes for CVode* and various constants */
60 #include "cvspgmr.h"           /* prototypes and constants for CVSPGMR solver */
61 #include "cvbbdpre.h"          /* prototypes for CVBBDPRE module             */
62 #include "nvector_parallel.h"  /* definition of type N_Vector and macro      */
63                                /* NV_DATA_P                                  */
64 #include "mpi.h"               /* MPI constants and types                    */
65
66
67 /* Problem Constants */
68
69 #define ZERO         RCONST(0.0)
70
71 #define NVARS        2                 /* number of species         */
72 #define KH           RCONST(4.0e-6)    /* horizontal diffusivity Kh */
73 #define VEL          RCONST(0.001)     /* advection velocity V      */
74 #define KV0          RCONST(1.0e-8)    /* coefficient in Kv(y)      */
75 #define Q1           RCONST(1.63e-16)  /* coefficients q1, q2, c3   */
76 #define Q2           RCONST(4.66e-16)
77 #define C3           RCONST(3.7e16)
78 #define A3           RCONST(22.62)     /* coefficient in expression for q3(t) */
79 #define A4           RCONST(7.601)     /* coefficient in expression for q4(t) */
80 #define C1_SCALE     RCONST(1.0e6)     /* coefficients in initial profiles    */
81 #define C2_SCALE     RCONST(1.0e12)
82
83 #define T0           ZERO                /* initial time */
84 #define NOUT         12                  /* number of output times */
85 #define TWOHR        RCONST(7200.0)      /* number of seconds in two hours  */
86 #define HALFDAY      RCONST(4.32e4)      /* number of seconds in a half day */
87 #define PI       RCONST(3.1415926535898) /* pi */
88
89 #define XMIN         ZERO                /* grid boundaries in x  */
90 #define XMAX         RCONST(20.0)
91 #define YMIN         RCONST(30.0)        /* grid boundaries in y  */
92 #define YMAX         RCONST(50.0)
93
94 #define NPEX         2             /* no. PEs in x direction of PE array */
95 #define NPEY         2             /* no. PEs in y direction of PE array */
96                                    /* Total no. PEs = NPEX*NPEY */
97 #define MXSUB        5             /* no. x points per subgrid */
98 #define MYSUB        5             /* no. y points per subgrid */
99
100 #define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
101 #define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
102                                     /* Spatial mesh is MX by MY */
103 /* CVodeMalloc Constants */
104
105 #define RTOL    RCONST(1.0e-5)    /* scalar relative tolerance */
106 #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 void f(realtype t, N_Vector u, N_Vector udot, void *f_data);

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

static void 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 ****************************/
```

```
161
162 int main(int argc, char *argv[])
163 {
164   UserData data;
165   void *cvode_mem;
166   void *pdata;
167   realtype abstol, reltol, t, tout;
168   N_Vector u;
169   int iout, my_pe, npes, flag, jpre;
170   long int neq, local_N, mudq, mldq, mukeep, mlkeep;
171   MPI_Comm comm;
172
173   data = NULL;
174   cvode_mem = pdata = NULL;
175   u = NULL;
176
177   /* Set problem size neq */
178   neq = NVARS*MX*MY;
179
180   /* Get processor number and total number of pe's */
181   MPI_Init(&argc, &argv);
182   comm = MPI_COMM_WORLD;
183   MPI_Comm_size(comm, &npes);
184   MPI_Comm_rank(comm, &my_pe);
185
186   if (npes != NPEX*NPEY) {
187     if (my_pe == 0)
188       fprintf(stderr, "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n\n",
189               npes, NPEX*NPEY);
190     MPI_Finalize();
191     return(1);
192   }
193
194   /* Set local length */
195   local_N = NVARS*MXSUB*MYSUB;
196
197   /* Allocate and load user data block */
198   data = (UserData) malloc(sizeof *data);
199   if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
200   InitUserData(my_pe, local_N, comm, data);
201
202   /* Allocate and initialize u, and set tolerances */
203   u = N_VNew_Parallel(comm, local_N, neq);
204   if(check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
205   SetInitialProfiles(u, data);
206   abstol = ATOL;
207   reltol = RTOL;
208
209   /*
210      Call CVodeCreate to create the solver memory:
211
212      CV_BDF     specifies the Backward Differentiation Formula
213      CV_NEWTON  specifies a Newton iteration
214
```

```
215       A pointer to the integrator memory is returned and stored in cvode_mem.
216    */
217
218    cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
219    if(check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
220
221    /* Set the pointer to user-defined data */
222    flag = CVodeSetFdata(cvode_mem, data);
223    if(check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
224
225    /*
226       Call CVodeMalloc to initialize the integrator memory:
227
228       cvode_mem is the pointer to the integrator memory returned by CVodeCreate
229       f       is the user's right hand side function in y'=f(t,y)
230       T0      is the initial time
231       u       is the initial dependent variable vector
232       CV_SS   specifies scalar relative and absolute tolerances
233       &reltol and &abstol are pointers to the scalar tolerances
234    */
235
236    flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
237    if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
238
239    /* Allocate preconditioner block */
240    mudq = mldq = NVARS*MXSUB;
241    mukeep = mlkeep = NVARS;
242    pdata = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
243                           mukeep, mlkeep, ZERO, flocal, NULL);
244    if(check_flag((void *)pdata, "CVBBDPrecAlloc", 0, my_pe)) MPI_Abort(comm, 1);
245
246    /* Call CVBBDSpgmr to specify the linear solver CVSPGMR using the
247       CVBBDPRE preconditioner, with left preconditioning and the
248       default maximum Krylov dimension maxl  */
249    flag = CVBBDSpgmr(cvode_mem, PREC_LEFT, 0, pdata);
250    if(check_flag(&flag, "CVBBDSpgmr", 1, my_pe)) MPI_Abort(comm, 1);
251
252    /* Print heading */
253    if (my_pe == 0) PrintIntro(npes, mudq, mldq, mukeep, mlkeep);
254
255    /* Loop over jpre (= PREC_LEFT, PREC_RIGHT), and solve the problem */
256    for (jpre = PREC_LEFT; jpre <= PREC_RIGHT; jpre++) {
257
258    /* On second run, re-initialize u, the integrator, CVBBDPRE, and CVSPGMR */
259
260    if (jpre == PREC_RIGHT) {
261
262      SetInitialProfiles(u, data);
263
264      flag = CVodeReInit(cvode_mem, f, T0, u, CV_SS, &reltol, &abstol);
265      if(check_flag(&flag, "CVodeReInit", 1, my_pe)) MPI_Abort(comm, 1);
266
267      flag = CVBBDPrecReInit(pdata, mudq, mldq, ZERO, flocal, NULL);
268      if(check_flag(&flag, "CVBBDPrecReInit", 1, my_pe)) MPI_Abort(comm, 1);
```

```
269
270       flag = CVSpgmrSetPrecType(cvode_mem, PREC_RIGHT);
271       check_flag(&flag, "CVSpgmrSetPrecType", 1, my_pe);
272
273       if (my_pe == 0) {
274         printf("\n\n-------------------------------------------------------------");
275         printf("------------\n");
276       }
277
278     }
279
280
281     if (my_pe == 0) {
282       printf("\n\nPreconditioner type is:  jpre = %s\n\n",
283              (jpre == PREC_LEFT) ? "PREC_LEFT" : "PREC_RIGHT");
284     }
285
286     /* In loop over output points, call CVode, print results, test for error */
287
288     for (iout = 1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
289       flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
290       if(check_flag(&flag, "CVode", 1, my_pe)) break;
291       PrintOutput(cvode_mem, my_pe, comm, u, t);
292     }
293
294     /* Print final statistics */
295
296     if (my_pe == 0) PrintFinalStats(cvode_mem, pdata);
297
298     } /* End of jpre loop */
299
300     /* Free memory */
301     N_VDestroy_Parallel(u);
302     CVBBDPrecFree(pdata);
303     free(data);
304     CVodeFree(cvode_mem);
305
306     MPI_Finalize();
307
308     return(0);
309   }
310
311   /*********************** Private Helper Functions ***********************/
312
313   /* Load constants in data */
314
315   static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
316                            UserData data)
317   {
318     int isubx, isuby;
319
320     /* Set problem constants */
321     data->om = PI/HALFDAY;
322     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
```

```
323   data->dy = (YMAX-YMIN)/((realtype)(MY-1));
324   data->hdco = KH/SQR(data->dx);
325   data->haco = VEL/(RCONST(2.0)*data->dx);
326   data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;
327
328   /* Set machine-related constants */
329   data->comm = comm;
330   data->my_pe = my_pe;
331   data->Nlocal = local_N;
332   /* isubx and isuby are the PE grid indices corresponding to my_pe */
333   isuby = my_pe/NPEX;
334   isubx = my_pe - isuby*NPEX;
335   data->isubx = isubx;
336   data->isuby = isuby;
337   /* Set the sizes of a boundary x-line in u and uext */
338   data->nvmxsub = NVARS*MXSUB;
339   data->nvmxsub2 = NVARS*(MXSUB+2);
340 }
341
342 /* Set initial conditions in u */
343
344 static void SetInitialProfiles(N_Vector u, UserData data)
345 {
346   int isubx, isuby;
347   int lx, ly, jx, jy;
348   long int offset;
349   realtype dx, dy, x, y, cx, cy, xmid, ymid;
350   realtype *uarray;
351
352   /* Set pointer to data array in vector u */
353
354   uarray = NV_DATA_P(u);
355
356   /* Get mesh spacings, and subgrid indices for this PE */
357
358   dx = data->dx;         dy = data->dy;
359   isubx = data->isubx;   isuby = data->isuby;
360
361   /* Load initial profiles of c1 and c2 into local u vector.
362   Here lx and ly are local mesh point indices on the local subgrid,
363   and jx and jy are the global mesh point indices. */
364
365   offset = 0;
366   xmid = RCONST(0.5)*(XMIN + XMAX);
367   ymid = RCONST(0.5)*(YMIN + YMAX);
368   for (ly = 0; ly < MYSUB; ly++) {
369     jy = ly + isuby*MYSUB;
370     y = YMIN + jy*dy;
371     cy = SQR(RCONST(0.1)*(y - ymid));
372     cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
373     for (lx = 0; lx < MXSUB; lx++) {
374       jx = lx + isubx*MXSUB;
375       x = XMIN + jx*dx;
376       cx = SQR(RCONST(0.1)*(x - xmid));
```

```
377         cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
378         uarray[offset  ] = C1_SCALE*cx*cy;
379         uarray[offset+1] = C2_SCALE*cx*cy;
380         offset = offset + 2;
381       }
382     }
383   }
384
385   /* Print problem introduction */
386
387   static void PrintIntro(int npes, long int mudq, long int mldq,
388                          long int mukeep, long int mlkeep)
389   {
390     printf("\n2-species diurnal advection-diffusion problem\n");
391     printf("  %d by %d mesh on %d processors\n", MX, MY, npes);
392     printf("  Using CVBBDPRE preconditioner module\n");
393     printf("    Difference-quotient half-bandwidths are");
394     printf(" mudq = %ld,  mldq = %ld\n", mudq, mldq);
395     printf("    Retained band block half-bandwidths are");
396     printf(" mukeep = %ld,  mlkeep = %ld", mukeep, mlkeep);
397
398     return;
399   }
400
401   /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
402
403   static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
404                           N_Vector u, realtype t)
405   {
406     int qu, flag, npelast;
407     long int i0, i1, nst;
408     realtype hu, *uarray, tempu[2];
409     MPI_Status status;
410
411     npelast = NPEX*NPEY - 1;
412     uarray = NV_DATA_P(u);
413
414     /* Send c1,c2 at top right mesh point to PE 0 */
415     if (my_pe == npelast) {
416       i0 = NVARS*MXSUB*MYSUB - 2;
417       i1 = i0 + 1;
418       if (npelast != 0)
419         MPI_Send(&uarray[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
420       else {
421         tempu[0] = uarray[i0];
422         tempu[1] = uarray[i1];
423       }
424     }
425
426     /* On PE 0, receive c1,c2 at top right, then print performance data
427        and sampled solution values */
428     if (my_pe == 0) {
429       if (npelast != 0)
430         MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
```

```
431       flag = CVodeGetNumSteps(cvode_mem, &nst);
432       check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
433       flag = CVodeGetLastOrder(cvode_mem, &qu);
434       check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
435       flag = CVodeGetLastStep(cvode_mem, &hu);
436       check_flag(&flag, "CVodeGetLastStep", 1, my_pe);
437   #if defined(SUNDIALS_EXTENDED_PRECISION)
438       printf("t = %.2Le   no. steps = %ld   order = %d   stepsize = %.2Le\n",
439              t, nst, qu, hu);
440       printf("At bottom left:  c1, c2 = %12.3Le %12.3Le \n", uarray[0], uarray[1]);
441       printf("At top right:    c1, c2 = %12.3Le %12.3Le \n\n", tempu[0], tempu[1]);
442   #elif defined(SUNDIALS_DOUBLE_PRECISION)
443       printf("t = %.2le   no. steps = %ld   order = %d   stepsize = %.2le\n",
444              t, nst, qu, hu);
445       printf("At bottom left:  c1, c2 = %12.3le %12.3le \n", uarray[0], uarray[1]);
446       printf("At top right:    c1, c2 = %12.3le %12.3le \n\n", tempu[0], tempu[1]);
447   #else
448       printf("t = %.2e   no. steps = %ld   order = %d   stepsize = %.2e\n",
449              t, nst, qu, hu);
450       printf("At bottom left:  c1, c2 = %12.3e %12.3e \n", uarray[0], uarray[1]);
451       printf("At top right:    c1, c2 = %12.3e %12.3e \n\n", tempu[0], tempu[1]);
452   #endif
453     }
454   }
455
456   /* Print final statistics contained in iopt */
457
458   static void PrintFinalStats(void *cvode_mem, void *pdata)
459   {
460     long int lenrw, leniw ;
461     long int lenrwSPGMR, leniwSPGMR;
462     long int lenrwBBDP, leniwBBDP, ngevalsBBDP;
463     long int nst, nfe, nsetups, nni, ncfn, netf;
464     long int nli, npe, nps, ncfl, nfeSPGMR;
465     int flag;
466
467     flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
468     check_flag(&flag, "CVodeGetWorkSpace", 1, 0);
469     flag = CVodeGetNumSteps(cvode_mem, &nst);
470     check_flag(&flag, "CVodeGetNumSteps", 1, 0);
471     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
472     check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
473     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
474     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
475     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
476     check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
477     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
478     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
479     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
480     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
481
482     flag = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
483     check_flag(&flag, "CVSpgmrGetWorkSpace", 1, 0);
484     flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
```

```
485   check_flag(&flag, "CVSpgmrGetNumLinIters", 1, 0);
486   flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
487   check_flag(&flag, "CVSpgmrGetNumPrecEvals", 1, 0);
488   flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
489   check_flag(&flag, "CVSpgmrGetNumPrecSolves", 1, 0);
490   flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
491   check_flag(&flag, "CVSpgmrGetNumConvFails", 1, 0);
492   flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);
493   check_flag(&flag, "CVSpgmrGetNumRhsEvals", 1, 0);
494
495   printf("\nFinal Statistics: \n\n");
496   printf("lenrw   = %5ld     leniw = %5ld\n", lenrw, leniw);
497   printf("llrw    = %5ld     lliw  = %5ld\n", lenrwSPGMR, leniwSPGMR);
498   printf("nst     = %5ld\n"                 , nst);
499   printf("nfe     = %5ld     nfel  = %5ld\n"  , nfe, nfeSPGMR);
500   printf("nni     = %5ld     nli   = %5ld\n"  , nni, nli);
501   printf("nsetups = %5ld     netf  = %5ld\n"  , nsetups, netf);
502   printf("npe     = %5ld     nps   = %5ld\n"  , npe, nps);
503   printf("ncfn    = %5ld     ncfl  = %5ld\n\n", ncfn, ncfl);
504
505   flag = CVBBDPrecGetWorkSpace(pdata, &lenrwBBDP, &leniwBBDP);
506   check_flag(&flag, "CVBBDPrecGetWorkSpace", 1, 0);
507   flag = CVBBDPrecGetNumGfnEvals(pdata, &ngevalsBBDP);
508   check_flag(&flag, "CVBBDPrecGetNumGfnEvals", 1, 0);
509   printf("In CVBBDPRE: real/integer local work space sizes = %ld, %ld\n",
510          lenrwBBDP, leniwBBDP);
511   printf("             no. flocal evals. = %ld\n",ngevalsBBDP);
512 }
513
514 /* Routine to send boundary data to neighboring PEs */
515
516 static void BSend(MPI_Comm comm,
517                   int my_pe, int isubx, int isuby,
518                   long int dsizex, long int dsizey,
519                   realtype uarray[])
520 {
521   int i, ly;
522   long int offsetu, offsetbuf;
523   realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
524
525   /* If isuby > 0, send data from bottom x-line of u */
526
527   if (isuby != 0)
528     MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
529
530   /* If isuby < NPEY-1, send data from top x-line of u */
531
532   if (isuby != NPEY-1) {
533     offsetu = (MYSUB-1)*dsizex;
534     MPI_Send(&uarray[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
535   }
536
537   /* If isubx > 0, send data from left y-line of u (via bufleft) */
538
```

```
539   if (isubx != 0) {
540     for (ly = 0; ly < MYSUB; ly++) {
541       offsetbuf = ly*NVARS;
542       offsetu = ly*dsizex;
543       for (i = 0; i < NVARS; i++)
544         bufleft[offsetbuf+i] = uarray[offsetu+i];
545     }
546     MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
547   }
548
549   /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
550
551   if (isubx != NPEX-1) {
552     for (ly = 0; ly < MYSUB; ly++) {
553       offsetbuf = ly*NVARS;
554       offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
555       for (i = 0; i < NVARS; i++)
556         bufright[offsetbuf+i] = uarray[offsetu+i];
557     }
558     MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
559   }
560
561 }
562
563 /* Routine to start receiving boundary data from neighboring PEs.
564    Notes:
565    1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
566    passed to both the BRecvPost and BRecvWait functions, and should not
567    be manipulated between the two calls.
568    2) request should have 4 entries, and should be passed in both calls also. */
569
570 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
571                       int my_pe, int isubx, int isuby,
572                       long int dsizex, long int dsizey,
573                       realtype uext[], realtype buffer[])
574 {
575   long int offsetue;
576   /* Have bufleft and bufright use the same buffer */
577   realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
578
579   /* If isuby > 0, receive data for bottom x-line of uext */
580   if (isuby != 0)
581     MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
582                                             my_pe-NPEX, 0, comm, &request[0]);
583
584   /* If isuby < NPEY-1, receive data for top x-line of uext */
585   if (isuby != NPEY-1) {
586     offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
587     MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
588                                          my_pe+NPEX, 0, comm, &request[1]);
589   }
590
591   /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
592   if (isubx != 0) {
```

```
593       MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
594                                           my_pe-1, 0, comm, &request[2]);
595     }
596
597     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
598     if (isubx != NPEX-1) {
599       MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
600                                           my_pe+1, 0, comm, &request[3]);
601     }
602
603   }
604
605   /* Routine to finish receiving boundary data from neighboring PEs.
606      Notes:
607      1) buffer should be able to hold 2*NVARS*MYSUB realtype entries, should be
608      passed to both the BRecvPost and BRecvWait functions, and should not
609      be manipulated between the two calls.
610      2) request should have 4 entries, and should be passed in both calls also. */
611
612   static void BRecvWait(MPI_Request request[],
613                         int isubx, int isuby,
614                         long int dsizex, realtype uext[],
615                         realtype buffer[])
616   {
617     int i, ly;
618     long int dsizex2, offsetue, offsetbuf;
619     realtype *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
620     MPI_Status status;
621
622     dsizex2 = dsizex + 2*NVARS;
623
624     /* If isuby > 0, receive data for bottom x-line of uext */
625     if (isuby != 0)
626       MPI_Wait(&request[0],&status);
627
628     /* If isuby < NPEY-1, receive data for top x-line of uext */
629     if (isuby != NPEY-1)
630       MPI_Wait(&request[1],&status);
631
632     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
633     if (isubx != 0) {
634       MPI_Wait(&request[2],&status);
635
636       /* Copy the buffer to uext */
637       for (ly = 0; ly < MYSUB; ly++) {
638         offsetbuf = ly*NVARS;
639         offsetue = (ly+1)*dsizex2;
640         for (i = 0; i < NVARS; i++)
641           uext[offsetue+i] = bufleft[offsetbuf+i];
642       }
643     }
644
645     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
646     if (isubx != NPEX-1) {
```

```
647       MPI_Wait(&request[3],&status);
648
649       /* Copy the buffer to uext */
650       for (ly = 0; ly < MYSUB; ly++) {
651         offsetbuf = ly*NVARS;
652         offsetue = (ly+2)*dsizex2 - NVARS;
653         for (i = 0; i < NVARS; i++)
654           uext[offsetue+i] = bufright[offsetbuf+i];
655       }
656     }
657   }
658
659   /* fucomm routine.  This routine performs all inter-processor
660      communication of data in u needed to calculate f.         */
661
662   static void fucomm(realtype t, N_Vector u, void *f_data)
663   {
664     UserData data;
665     realtype *uarray, *uext, buffer[2*NVARS*MYSUB];
666     MPI_Comm comm;
667     int my_pe, isubx, isuby;
668     long int nvmxsub, nvmysub;
669     MPI_Request request[4];
670
671     data = (UserData) f_data;
672     uarray = NV_DATA_P(u);
673
674     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
675
676     comm = data->comm;  my_pe = data->my_pe;
677     isubx = data->isubx;   isuby = data->isuby;
678     nvmxsub = data->nvmxsub;
679     nvmysub = NVARS*MYSUB;
680     uext = data->uext;
681
682     /* Start receiving boundary data from neighboring PEs */
683
684     BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);
685
686     /* Send data from boundary of local grid to neighboring PEs */
687
688     BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, uarray);
689
690     /* Finish receiving boundary data from neighboring PEs */
691
692     BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
693   }
694
695   /***************** Function called by the solver *************************/
696
697   /* f routine.  Evaluate f(t,y).  First call fucomm to do communication of
698      subgrid boundary data into uext.  Then calculate f by a call to flocal. */
699
700   static void 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);
}

/***************** 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 void 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;
```

```
809   /* Loop over all grid points in local subgrid */
810
811   for (ly = 0; ly < MYSUB; ly++) {
812
813     jy = ly + isuby*MYSUB;
814
815     /* Set vertical diffusion coefficients at jy +- 1/2 */
816
817     ydn = YMIN + (jy - RCONST(0.5))*dely;
818     yup = ydn + dely;
819     cydn = verdco*exp(RCONST(0.2)*ydn);
820     cyup = verdco*exp(RCONST(0.2)*yup);
821     for (lx = 0; lx < MXSUB; lx++) {
822
823       jx = lx + isubx*MXSUB;
824
825       /* Extract c1 and c2, and set kinetic rate terms */
826
827       offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
828       c1 = uext[offsetue];
829       c2 = uext[offsetue+1];
830       qq1 = Q1*c1*C3;
831       qq2 = Q2*c1*c2;
832       qq3 = q3*C3;
833       qq4 = q4coef*c2;
834       rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
835       rkin2 = qq1 - qq2 - qq4;
836
837       /* Set vertical diffusion terms */
838
839       c1dn = uext[offsetue-nvmxsub2];
840       c2dn = uext[offsetue-nvmxsub2+1];
841       c1up = uext[offsetue+nvmxsub2];
842       c2up = uext[offsetue+nvmxsub2+1];
843       vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
844       vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
845
846       /* Set horizontal diffusion and advection terms */
847
848       c1lt = uext[offsetue-2];
849       c2lt = uext[offsetue-1];
850       c1rt = uext[offsetue+2];
851       c2rt = uext[offsetue+3];
852       hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
853       hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
854       horad1 = horaco*(c1rt - c1lt);
855       horad2 = horaco*(c2rt - c2lt);
856
857       /* Load all terms into duarray */
858
859       offsetu = lx*NVARS + ly*nvmxsub;
860       duarray[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
861       duarray[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
862     }
```

```
  }
}

/* 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 = 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 cvkryf.f

```
1  C     -----------------------------------------------------------------
2  C     $Revision: 1.20 $
3  C     $Date: 2004/11/22 23:23:27 $
4  C     -----------------------------------------------------------------
5  C     FCVODE Example Problem: 2D kinetics-transport, precond. Krylov
6  C     solver.
7  C
8  C     An ODE system is generated from the following 2-species diurnal
9  C     kinetics advection-diffusion PDE system in 2 space dimensions:
10 C
11 C     dc(i)/dt = Kh*(d/dx)**2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)
12 C                           + Ri(c1,c2,t)      for i = 1,2,   where
13 C     R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,
14 C     R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,
15 C     Kv(y) = Kv0*exp(y/5) ,
16 C     Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)
17 C     vary diurnally.
18 C
19 C     The problem is posed on the square
20 C     0 .le. x .le. 20,    30 .le. y .le. 50   (all in km),
21 C     with homogeneous Neumann boundary conditions, and for time t
22 C     in 0 .le. t .le. 86400 sec (1 day).
23 C     The PDE system is treated by central differences on a uniform
24 C     10 x 10 mesh, with simple polynomial initial profiles.
25 C     The problem is solved with CVODE, with the BDF/GMRES method and
26 C     the block-diagonal part of the Jacobian as a left
27 C     preconditioner.
28 C
29 C     Note: this program requires the dense linear solver routines
30 C     DGEFA and DGESL from LINPACK, and BLAS routines DCOPY and DSCAL.
31 C
32 C     The second and third dimensions of U here must match the values
33 C     of MESHX and MESHY, for consistency with the output statements
34 C     below.
35 C     -----------------------------------------------------------------
36 C
37       IMPLICIT NONE
38 C
39       INTEGER METH, ITMETH, IATOL, INOPT, ITASK, IER, LNCFL, LNPS
40       INTEGER LNST, LNFE, LNSETUP, LNNI, LNCF, LQ, LH, LNPE, LNLI
41       INTEGER IOUT, JPRETYPE, IGSTYPE, MAXL
42       INTEGER*4 IOPT(40)
43       INTEGER*4 NEQ, MESHX, MESHY, NST, NFE, NPSET, NPE, NPS, NNI
44       INTEGER*4 NLI, NCFN, NCFL
45       DOUBLE PRECISION ATOL, AVDIM, T, TOUT, TWOHR, RTOL, FLOOR, DELT
46       DOUBLE PRECISION U(2,10,10), ROPT(40)
47 C
48       DATA TWOHR/7200.0D0/, RTOL/1.0D-5/, FLOOR/100.0D0/,
49      1     JPRETYPE/1/, IGSTYPE/1/, MAXL/0/, DELT/0.0D0/
50       DATA LNST/4/, LNFE/5/, LNSETUP/6/, LNNI/7/, LNCF/8/,
51      1     LQ/11/, LH/5/, LNPE/18/, LNLI/19/, LNPS/20/, LNCFL/21/
52       COMMON /PBDIM/ NEQ
```

```
C
C Set mesh sizes
      MESHX = 10
      MESHY = 10
C Load Common and initial values in Subroutine INITKX
      CALL INITKX(MESHX, MESHY, U)
C Set other input arguments.
      NEQ = 2 * MESHX * MESHY
      T = 0.0D0
      METH = 2
      ITMETH = 2
      IATOL = 1
      ATOL = RTOL * FLOOR
      INOPT = 0
      ITASK = 1
C
      WRITE(6,10) NEQ
 10   FORMAT('Krylov example problem:'//
     1       ' Kinetics-transport, NEQ = ', I4/)
C
      CALL FNVINITS(NEQ, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,20) IER
 20     FORMAT(///' SUNDIALS_ERROR: FNVINITS returned IER = ', I5)
        STOP
      ENDIF
C
      CALL FCVMALLOC(T, U, METH, ITMETH, IATOL, RTOL, ATOL,
     1               INOPT, IOPT, ROPT, IER)
      IF (IER .NE. 0) THEN
        WRITE(6,30) IER
 30     FORMAT(///' SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
        CALL FNVFREES
        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 FNVFREES
        CALL FCVFREE
        STOP
      ENDIF
C
      CALL FCVSPGMRSETPSET(1, IER)
C
      CALL FCVSPGMRSETPSOL(1, IER)
C
C Loop over output points, call FCVODE, print sample solution values.
      TOUT = TWOHR
      DO 70 IOUT = 1, 12
C
        CALL FCVODE(TOUT, T, U, ITASK, IER)
```

```
C
        WRITE(6,50) T, IOPT(LNST), IOPT(LQ), ROPT(LH)
 50     FORMAT(/' t = ', E11.3, 5X, 'no. steps = ', I5,
     1         '   order = ', I3, '   stepsize = ', E14.6)
        WRITE(6,55) U(1,1,1), U(1,5,5), U(1,10,10),
     1             U(2,1,1), U(2,5,5), U(2,10,10)
 55     FORMAT('  c1 (bot.left/middle/top rt.) = ', 3E14.6/
     1         '  c2 (bot.left/middle/top rt.) = ', 3E14.6)
C
        IF (IER .NE. 0) THEN
          WRITE(6,60) IER, IOPT(26)
 60       FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
     1           '                 Linear Solver returned IER = ', I5)
          CALL FNVFREES
          CALL FCVFREE
          STOP
          ENDIF
C
        TOUT = TOUT + TWOHR
 70     CONTINUE

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

      SUBROUTINE INITKX(MESHX, MESHY, U0)
C Routine to set problem constants and initial values
C
```

```
161         IMPLICIT NONE
162   C
163         INTEGER*4 MESHX, MESHY
164         INTEGER*4 MX, MY, MM, JY, JX, NEQ
165         DOUBLE PRECISION U0
166         DIMENSION U0(2,MESHX,MESHY)
167         DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
168         DOUBLE PRECISION VDCO, HACO, X, Y
169         DOUBLE PRECISION CX, CY, DKH, DKV0, DX, HALFDA, PI, VEL
170   C
171         COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
172         COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
173         DATA DKH/4.0D-6/, VEL/0.001D0/, DKV0/1.0D-8/, HALFDA/4.32D4/,
174        1     PI/3.1415926535898D0/
175   C
176   C Load Common block of problem parameters.
177         MX = MESHX
178         MY = MESHY
179         MM = MX * MY
180         NEQ = 2 * MM
181         Q1 = 1.63D-16
182         Q2 = 4.66D-16
183         A3 = 22.62D0
184         A4 = 7.601D0
185         OM = PI / HALFDA
186         C3 = 3.7D16
187         DX = 20.0D0 / (MX - 1.0D0)
188         DY = 20.0D0 / (MY - 1.0D0)
189         HDCO = DKH / DX**2
190         HACO = VEL / (2.0D0 * DX)
191         VDCO = (1.0D0 / DY**2) * DKV0
192   C
193   C Set initial profiles.
194         DO 20 JY = 1, MY
195           Y = 30.0D0 + (JY - 1.0D0) * DY
196           CY = (0.1D0 * (Y - 40.0D0))**2
197           CY = 1.0D0 - CY + 0.5D0 * CY**2
198           DO 10 JX = 1, MX
199             X = (JX - 1.0D0) * DX
200             CX = (0.1D0 * (X - 10.0D0))**2
201             CX = 1.0D0 - CX + 0.5D0 * CX**2
202             U0(1,JX,JY) = 1.0D6 * CX * CY
203             U0(2,JX,JY) = 1.0D12 * CX * CY
204    10       CONTINUE
205    20     CONTINUE
206   C
207         RETURN
208         END
209
210         SUBROUTINE FCVFUN(T, U, UDOT)
211   C Routine for right-hand side function f
212   C
213         IMPLICIT NONE
214   C
```

```
      INTEGER ILEFT, IRIGHT
      INTEGER*4 JX, JY, MX, MY, MM, IBLOK0, IBLOK, IDN, IUP
      DOUBLE PRECISION T, U(2,*), UDOT(2,*)
      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
      COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
      COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
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
C
C Loop over all grid points.
      DO 20 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 10 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
 10        CONTINUE
 20      CONTINUE
       RETURN
       END

       SUBROUTINE FCVPSET(T, U, FU, JOK, JCUR, GAMMA, EWT, H,
      1                   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, H
       INTEGER*4 LENBD, JY, JX, IBLOK, MX, MY, MM
       INTEGER*4 IBLOK0, IPP
       DOUBLE PRECISION T, U(2,*), GAMMA
       DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
       DOUBLE PRECISION VDCO, HACO
       DOUBLE PRECISION BD, P, FU, EWT, V1, V2, V3
       DOUBLE PRECISION C1, C2, CYDN, CYUP, DIAG, TEMP, YDN, YUP
C
       COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
       COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
       COMMON /BDJ/ BD(2,2,100), P(2,2,100), IPP(2,100)
C
       IER = 0
       LENBD = 4 * MM
C
C If JOK = 1, copy BD to P.
       IF (JOK .EQ. 1) THEN
         CALL DCOPY(LENBD, BD(1,1,1), 1, P(1,1,1), 1)
         JCUR = 0
       ELSE
C
C JOK = 0.  Compute diagonal Jacobian blocks and copy to P.
C   (using q4 value computed on last FCVFUN call).
       DO 20 JY = 1, MY
         YDN = 30.0D0 + (JY - 1.5D0) * DY
         YUP = YDN + DY
         CYDN = VDCO * EXP(0.2D0 * YDN)
```

```
323           CYUP = VDCO * EXP(0.2D0 * YUP)
324           DIAG = -(CYDN + CYUP + 2.0D0 * HDCO)
325           IBLOK0 = (JY - 1) * MX
326           DO 10 JX = 1, MX
327             IBLOK = IBLOK0 + JX
328             C1 = U(1,IBLOK)
329             C2 = U(2,IBLOK)
330             BD(1,1,IBLOK) = (-Q1 * C3 - Q2 * C2) + DIAG
331             BD(1,2,IBLOK) = -Q2 * C1 + Q4
332             BD(2,1,IBLOK) =  Q1 * C3 - Q2 * C2
333             BD(2,2,IBLOK) = (-Q2 * C1 - Q4) + DIAG
334    10       CONTINUE
335    20     CONTINUE
336         CALL DCOPY(LENBD, BD(1,1,1), 1, P(1,1,1), 1)
337         JCUR = 1
338         ENDIF
339   C
340   C Scale P by -GAMMA.
341         TEMP = -GAMMA
342         CALL DSCAL(LENBD, TEMP, P, 1)
343   C
344   C Add identity matrix and do LU decompositions on blocks, in place.
345         DO 40 IBLOK = 1, MM
346           P(1,1,IBLOK) = P(1,1,IBLOK) + 1.0D0
347           P(2,2,IBLOK) = P(2,2,IBLOK) + 1.0D0
348           CALL DGEFA(P(1,1,IBLOK), 2, 2, IPP(1,IBLOK), IER)
349           IF (IER .NE. 0) RETURN
350    40     CONTINUE
351   C
352         RETURN
353         END
354
355         SUBROUTINE FCVPSOL(T, U, FU, VTEMP, GAMMA, EWT, DELTA,
356        1                   R, LR, Z, IER)
357   C Routine to solve preconditioner linear system.
358   C Note: The dimensions in /BDJ/ below assume at most 100 mesh points.
359   C
360         IMPLICIT NONE
361   C
362         INTEGER IER
363         INTEGER*4 I, NEQ, MX, MY, MM, LR, IPP
364         DOUBLE PRECISION R(*), Z(2,*)
365         DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
366         DOUBLE PRECISION VDCO, HACO
367         DOUBLE PRECISION BD, P, T, U, FU, VTEMP, EWT, DELTA, GAMMA
368   C
369         COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
370         COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
371         COMMON /BDJ/ BD(2,2,100), P(2,2,100), IPP(2,100)
372         COMMON /PBDIM/ NEQ
373   C
374   C Solve the block-diagonal system Px = r using LU factors stored in P
375   C and pivot data in IPP, and return the solution in Z.
376         IER = 0
```

```
377         CALL DCOPY(NEQ, R, 1, Z, 1)
378         DO 10 I = 1, MM
379           CALL DGESL(P(1,1,I), 2, 2, IPP(1,I), Z(1,I), 0)
380    10     CONTINUE
381         RETURN
382         END
383
384         subroutine dgefa(a, lda, n, ipvt, info)
385   c
386         implicit none
387   c
388         integer info, idamax, j, k, kp1, l, nm1, n
389         integer*4 lda, ipvt(1)
390         double precision a(lda,1), t
391   c
392   c     dgefa factors a double precision matrix by gaussian elimination.
393   c
394   c     dgefa is usually called by dgeco, but it can be called
395   c     directly with a saving in time if  rcond  is not needed.
396   c     (time for dgeco) = (1 + 9/n)*(time for dgefa) .
397   c
398   c     on entry
399   c
400   c        a       double precision(lda, n)
401   c                the matrix to be factored.
402   c
403   c        lda     integer
404   c                the leading dimension of the array  a .
405   c
406   c        n       integer
407   c                the order of the matrix  a .
408   c
409   c     on return
410   c
411   c        a       an upper triangular matrix and the multipliers
412   c                which were used to obtain it.
413   c                the factorization can be written  a = l*u  where
414   c                l  is a product of permutation and unit lower
415   c                triangular matrices and  u  is upper triangular.
416   c
417   c        ipvt    integer(n)
418   c                an integer vector of pivot indices.
419   c
420   c        info    integer
421   c                = 0  normal value.
422   c                = k  if  u(k,k) .eq. 0.0 .  this is not an error
423   c                     condition for this subroutine, but it does
424   c                     indicate that dgesl or dgedi will divide by zero
425   c                     if called.  use  rcond  in dgeco for a reliable
426   c                     indication of singularity.
427   c
428   c     linpack. this version dated 08/14/78 .
429   c     cleve moler, university of new mexico, argonne national lab.
430   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
```

```
593               k = n - kb
594               b(k) = b(k) + ddot(n - k, a(k + 1,k), 1, b(k + 1), 1)
595               l = ipvt(k)
596               if (l .eq. k) go to 70
597                  t = b(l)
598                  b(l) = b(k)
599                  b(k) = t
600      70       continue
601      80    continue
602      90    continue
603     100 continue
604         return
605         end
606   c
607         subroutine daxpy(n, da, dx, incx, dy, incy)
608   c
609   c     constant times a vector plus a vector.
610   c     uses unrolled loops for increments equal to one.
611   c     jack dongarra, linpack, 3/11/78.
612   c
613         implicit none
614   c
615         integer i, incx, incy, ix, iy, m, mp1
616         integer*4 n
617         double precision dx(1), dy(1), da
618   c
619         if (n .le. 0) return
620         if (da .eq. 0.0d0) return
621         if (incx .eq. 1 .and. incy .eq. 1) go to 20
622   c
623   c        code for unequal increments or equal increments
624   c        not equal to 1
625   c
626         ix = 1
627         iy = 1
628         if (incx .lt. 0) ix = (-n + 1) * incx + 1
629         if (incy .lt. 0) iy = (-n + 1) * incy + 1
630         do 10 i = 1, n
631           dy(iy) = dy(iy) + da * dx(ix)
632           ix = ix + incx
633           iy = iy + incy
634      10 continue
635         return
636   c
637   c        code for both increments equal to 1
638   c
639   c
640   c        clean-up loop
641   c
642      20 m = mod(n, 4)
643         if ( m .eq. 0 ) go to 40
644         do 30 i = 1, m
645           dy(i) = dy(i) + da * dx(i)
646      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
```

```
701         end
702   c
703         double precision function ddot(n, dx, incx, dy, incy)
704   c
705   c     forms the dot product of two vectors.
706   c     uses unrolled loops for increments equal to one.
707   c     jack dongarra, linpack, 3/11/78.
708   c
709         implicit none
710   c
711         integer i, incx, incy, ix, iy, m, mp1
712         integer*4 n
713         double precision dx(1), dy(1), dtemp
714   c
715         ddot = 0.0d0
716         dtemp = 0.0d0
717         if (n .le. 0) return
718         if (incx .eq. 1 .and. incy .eq. 1) go to 20
719   c
720   c        code for unequal increments or equal increments
721   c          not equal to 1
722   c
723         ix = 1
724         iy = 1
725         if (incx .lt. 0) ix = (-n + 1) * incx + 1
726         if (incy .lt. 0) iy = (-n + 1) * incy + 1
727         do 10 i = 1, n
728           dtemp = dtemp + dx(ix) * dy(iy)
729           ix = ix + incx
730           iy = iy + incy
731      10 continue
732         ddot = dtemp
733         return
734   c
735   c        code for both increments equal to 1
736   c
737   c
738   c        clean-up loop
739   c
740      20 m = mod(n, 5)
741         if ( m .eq. 0 ) go to 40
742         do 30 i = 1,m
743           dtemp = dtemp + dx(i) * dy(i)
744      30 continue
745         if ( n .lt. 5 ) go to 60
746      40 mp1 = m + 1
747         do 50 i = mp1, n, 5
748           dtemp = dtemp + dx(i) * dy(i) + dx(i + 1) * dy(i + 1) +
749       *          dx(i + 2) * dy(i + 2) + dx(i + 3) * dy(i + 3) +
750       *          dx(i + 4) * dy(i + 4)
751      50 continue
752      60 ddot = dtemp
753         return
754         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 pvdiagkbf.f

```
C     -----------------------------------------------------------------
C     $Revision: 1.18 $
C     $Date: 2004/10/21 18:58:44 $
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.

      INCLUDE "mpif.h"
C
      INTEGER NOUT, LNST, LNFE, LNSETUP, LNNI, LNCF, LNETF, LNPE
      INTEGER LNLI, LNPS, LNCFL, MYPE, IER, NPES, METH, ITMETH
      INTEGER IATOL, INOPT, ITASK, IPRE, IGS, IOUT
      INTEGER*4 IOPT(40)
      INTEGER*4 NEQ, NLOCAL, I, MUDQ, MLDQ, MU, ML, NETF
      INTEGER*4 NST, NFE, NPSET, NPE, NPS, NNI, NLI, NCFN, NCFL
      INTEGER*4 LENRPW, LENIPW, NGE
      DOUBLE PRECISION ALPHA, TOUT, ERMAX, AVDIM
      DOUBLE PRECISION ATOL, ERRI, RTOL, GERMAX, DTOUT, Y, ROPT, T
      DIMENSION Y(1024), ROPT(40)
C
      DATA ATOL/1.0D-10/, RTOL/1.0D-5/, DTOUT/0.1D0/, NOUT/10/
      DATA LNST/4/, LNFE/5/, LNSETUP/6/, LNNI/7/, LNCF/8/, LNETF/9/,
     1     LNPE/18/, LNLI/19/, LNPS/20/, LNCFL/21/
C
      COMMON /PCOM/ ALPHA, NLOCAL, MYPE
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.
      NLOCAL = 10
      NEQ = NPES * NLOCAL
      T = 0.0D0
      METH = 2
      ITMETH = 2
      IATOL = 1
      INOPT = 0
      ITASK = 1
      IPRE = 1
      IGS = 1
C     Set parameter alpha
      ALPHA  = 10.0D0
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(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,
     &               INOPT, IOPT, ROPT, 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 IOUT = 1, NOUT
C
         CALL FCVODE(TOUT, T, Y, ITASK, IER)
C
         IF (MYPE .EQ. 0) WRITE(6,45) T, IOPT(LNST), IOPT(LNFE)
 45      FORMAT(' t = ', E10.2, 5X, 'no. steps = ', I5,
     &          '   no. f-s = ', I5)
C
         IF (IER .NE. 0) THEN
            WRITE(6,50) IER, IOPT(26)
 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 = IOPT(LNST)
         NFE = IOPT(LNFE)
         NPSET = IOPT(LNSETUP)
         NPE = IOPT(LNPE)
         NPS = IOPT(LNPS)
         NNI = IOPT(LNNI)
         NLI = IOPT(LNLI)
         AVDIM = DBLE(NLI) / DBLE(NNI)
         NCFN = IOPT(LNCF)
         NCFL = IOPT(LNCFL)
         NETF = IOPT(LNETF)
         WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
     &               NCFL, NETF
 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/)
         CALL FCVBBDOPT(LENRPW, LENIPW, NGE)
         WRITE(6,82) LENRPW, LENIPW, NGE
 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 case IPRE = 2.
C     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,INOPT, IOPT, ROPT, IER)
       IF (IER .NE. 0) THEN
         WRITE(6,91) IER
 91      FORMAT(///' SUNDIALS_ERROR: FCVREINIT returned IER = ', I5)
```

```
215           CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
216           STOP
217        ENDIF
218  C
219        IPRE = 2
220  C
221        CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, 0.0D0, IER)
222        IF (IER .NE. 0) THEN
223           WRITE(6,92) IER
224   92      FORMAT(///' SUNDIALS_ERROR: FCVBBDREINIT returned IER = ', I5)
225           CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
226           STOP
227        ENDIF
228  C
229        CALL FCVSPGMRREINIT(IPRE, IGS, 0.0D0, IER)
230        IF (IER .NE. 0) THEN
231           WRITE(6,93) IER
232   93      FORMAT(///' SUNDIALS_ERROR: FCVSPGMRREINIT returned IER = ',I5)
233           CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
234           STOP
235        ENDIF
236  C
237        IF (MYPE .EQ. 0) WRITE(6,95)
238   95   FORMAT(//60('-')///'Preconditioning on right'/)
239        GO TO 40
240  C
241  C     Free the memory and finalize MPI.
242   99   CALL FCVBBDFREE
243        CALL FCVFREE
244        CALL FNVFREEP
245        CALL MPI_FINALIZE(IER)
246  C
247        STOP
248        END
249  C
250        SUBROUTINE FCVFUN(T, Y, YDOT)
251  C     Routine for right-hand side function f
252  C
253        IMPLICIT NONE
254  C
255        INTEGER MYPE
256        INTEGER*4 I, NLOCAL
257        DOUBLE PRECISION Y, YDOT, ALPHA, T
258        DIMENSION Y(*), YDOT(*)
259  C
260        COMMON /PCOM/ ALPHA, NLOCAL, MYPE
261  C
262        DO I = 1, NLOCAL
263           YDOT(I) = -ALPHA * (MYPE * NLOCAL + I) * Y(I)
264        ENDDO
265  C
266        RETURN
267        END
268  C
```

```
      SUBROUTINE FCVGLOCFN(NLOC, T, YLOC, GLOC)
C     Routine to define local approximate function g, here the same as f.
      IMPLICIT NONE
C
      INTEGER*4 NLOC
      DOUBLE PRECISION YLOC, GLOC, T
      DIMENSION YLOC(*), GLOC(*)
C
      CALL FCVFUN(T, YLOC, GLOC)
C
      RETURN
      END

      SUBROUTINE FCVCOMMFN(NLOC, T, YLOC)
C     Routine to perform communication required for evaluation of g.
      RETURN
      END
```

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551