

# Example Programs for CVODE v2.3.0

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

April 2005

## **DISCLAIMER**

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

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Serial example problems</b>	<b>4</b>
2.1	A dense example: <code>cvdx</code> . . . . .	4
2.2	A banded example: <code>cvbx</code> . . . . .	6
2.3	A Krylov example: <code>cvkx</code> . . . . .	9
<b>3</b>	<b>Parallel example problems</b>	<b>13</b>
3.1	A nonstiff example: <code>pvnx</code> . . . . .	13
3.2	A user preconditioner example: <code>pvkx</code> . . . . .	15
3.3	A CVBBDPRE preconditioner example: <code>pvkxb</code> . . . . .	17
<b>4</b>	<b>Fortran example problems</b>	<b>21</b>
4.1	A serial example: <code>cvkryf</code> . . . . .	21
4.2	A parallel example: <code>pvdiagkbf</code> . . . . .	23
<b>5</b>	<b>Parallel tests</b>	<b>25</b>
	<b>References</b>	<b>27</b>
<b>A</b>	<b>Listing of <code>cvdx.c</code></b>	<b>28</b>
<b>B</b>	<b>Listing of <code>cvbx.c</code></b>	<b>35</b>
<b>C</b>	<b>Listing of <code>cvkx.c</code></b>	<b>44</b>
<b>D</b>	<b>Listing of <code>pvnx.c</code></b>	<b>57</b>
<b>E</b>	<b>Listing of <code>pvkx.c</code></b>	<b>64</b>
<b>F</b>	<b>Listing of <code>pvkxb.c</code></b>	<b>83</b>
<b>G</b>	<b>Listing of <code>cvkryf.f</code></b>	<b>100</b>
<b>H</b>	<b>Listing of <code>pvdiagkbf.f</code></b>	<b>116</b>



# 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 `CVDENLSE` 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.
- `cvdxe` is the same as `cvdx` but demonstrates the user-supplied error weight function feature of CVODE.
- `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 preconditioner 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/fcmix/examples_ser` and `sundials/cvode/fcmix/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. 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 §5.6.3.

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

The `main` program begins with some dimensions and type declarations, including use of the 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 dependent variable vector into `y` and the absolute tolerances into `abstol` using the `Ith` macro.

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

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

The call to `CVodeMalloc` allocates the solver memory block. Its arguments include the name of the C function `f` defining the right-hand side function  $f(t, y)$ , and the initial values of  $t$  and  $y$ . The argument `CV_SV` specifies a vector of absolute tolerances, and this is followed by the value of the relative tolerance `reltol` and the absolute tolerance vector `abstol`. See §5.5.1 for full details of this call.

The call to `CVodeRootInit` specifies that a rootfinding problem is to be solved along with the integration of the ODE system, that the root functions are specified in the function `g`, and that there are two such functions. Specifically, they are set to  $y_1 - 0.0001$  and  $y_3 - 0.01$ , respectively. See §5.7.1 for a detailed description of this call.

The calls to `CVDense` (see §5.5.2) 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  $t = \text{tout}$ , putting the computed value of  $y(\text{tout})$  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 §5.6.3 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    ncnfn = 0    netf = 26    nge = 543

```

## 2.2 A banded example: `cvbx`

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

$$\partial v / \partial t = \partial^2 v / \partial x^2 + .5 \partial v / \partial x + \partial^2 v / \partial y^2 \quad (2)$$

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

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

where  $1 \leq i \leq \text{MX}$  and  $1 \leq j \leq \text{MY}$ . The boundary conditions are imposed by taking  $v_{ij} = 0$  above if  $i = 0$  or  $\text{MX}+1$ , or if  $j = 0$  or  $\text{MY}+1$ . If we set  $u_{(j-1)+i-1*\text{MY}} = v_{ij}$ , 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  $\text{MY}$ . In the example, we take  $\text{MX} = 10$  and  $\text{MY} = 5$ . 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,  $\text{MX}$  and  $\text{MY}$ , 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  $\text{MY}$ . The call to `CVBandSetJacFn` (see §5.5.4) specifies that a user-supplied Jacobian function `Jac` is to be used and that a pointer to `data` should be passed to `Jac` every time it is called. The actual solution of the problem is performed by the call to `CVode` within the loop over the output times `tout`. The max-norm of the solution vector (from a call to `N_VMaxNorm`) and the cumulative number of time steps (from a call to `CVodeGetNumSteps`) are printed at each output time. Finally, the calls to `PrintFinalStats`, `N_VDestroy`, and `CVodeFree` print statistics and free problem memory.

Following the `main` program in the `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 Jacobian 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 `uodot`. Note the extra lines setting out-of-bounds values of  $u$  to zero.

The `Jac` function is an expression of the derivatives

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

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 §5.6.4 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

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

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

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

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

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

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

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

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

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

Among the initial `#include` lines in this case are lines to include `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 `realtyp` 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 `max1`. The Gram-Schmidt orthogonalization is set to `MODIFIED_GS` through the function `CVSpgmrSetGSType`. Next, user-supplied preconditioner setup and solve functions, `Precond` and `PSolve`, as well as the `data` pointer passed to `Precond` and `PSolve` whenever these are called, See §5.5.4 for details on the `CVSpgmrSetPreconditioner` function.

Then for a sequence of `tout` values, `CVode` is called in the `CV_NORMAL` mode, sampled output is printed, and the return value is tested for error conditions. After that, `PrintFinalStats` is called to get and print final statistics, and memory is freed by calls to `N_VDestroy`, `FreeUserData`, and `CVodeFree`. The printed statistics include various counters, such as the total numbers of steps (`nst`), of `f` evaluations (excluding those for  $Jv$  product evaluations) (`nfe`), of `f` evaluations for  $Jv$  evaluations (`nfel`), of nonlinear iterations (`nni`), of linear (Krylov) iterations (`nli`), of preconditioner setups (`nsetups`), of preconditioner evaluations (`npe`), and of preconditioner solves (`nps`), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator `CVode`, denoted `lenrw` and `leniw`, and those used by `CVSPGMR`, denoted `llrw` and `lliw`. All of these optional outputs are described in §5.5.6. 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 §5.6.7) and backsolved in the `PSolve` routine (see §5.6.6). Its diagonal blocks are  $2 \times 2$  matrices that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian 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  $2 \times 2$  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 retrieve

and print selected solution values and statistics; `PrintFinalStats` to print statistics; and `check_flag` to check return values for error conditions.

The output generated by `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 = 301  order = 5  stepsize = 3.72e+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 = 329  order = 5  stepsize = 8.62e+01
c1 (bot.left/middle/top rt.) = 1.404e+04  2.029e+04  1.561e+04
c2 (bot.left/middle/top rt.) = 3.387e+11  4.894e+11  3.765e+11

t = 4.32e+04  no. steps = 386  order = 4  stepsize = 4.03e+02
c1 (bot.left/middle/top rt.) = -2.083e-07  -6.285e-07  -2.237e-07
c2 (bot.left/middle/top rt.) = 3.382e+11  1.355e+11  3.804e+11

t = 5.04e+04  no. steps = 399  order = 5  stepsize = 4.22e+02
c1 (bot.left/middle/top rt.) = -5.968e-09  5.891e-07  -9.151e-09
c2 (bot.left/middle/top rt.) = 3.358e+11  4.930e+11  3.864e+11

t = 5.76e+04  no. steps = 416  order = 4  stepsize = 1.05e+02
c1 (bot.left/middle/top rt.) = 8.838e-08  -1.508e-06  1.409e-07
c2 (bot.left/middle/top rt.) = 3.320e+11  9.650e+11  3.909e+11

t = 6.48e+04  no. steps = 432  order = 4  stepsize = 5.14e+02
c1 (bot.left/middle/top rt.) = 7.999e-11  -2.155e-09  1.308e-10
c2 (bot.left/middle/top rt.) = 3.313e+11  8.922e+11  3.963e+11

t = 7.20e+04  no. steps = 446  order = 4  stepsize = 5.14e+02
c1 (bot.left/middle/top rt.) = 7.272e-15  -1.817e-13  1.188e-14
c2 (bot.left/middle/top rt.) = 3.330e+11  6.186e+11  4.039e+11

t = 7.92e+04  no. steps = 460  order = 4  stepsize = 5.14e+02
c1 (bot.left/middle/top rt.) = 4.110e-18  -2.359e-14  6.131e-18
c2 (bot.left/middle/top rt.) = 3.334e+11  6.669e+11  4.120e+11

```

```
t = 8.64e+04   no. steps = 474   order = 4   stepsize = 5.14e+02
c1 (bot.left/middle/top rt.) =   7.647e-19   1.346e-14   -1.473e-17
c2 (bot.left/middle/top rt.) =   3.352e+11   9.108e+11   4.163e+11
```

Final Statistics..

```
lenrw = 2000   leniw = 10
llrw  = 2046   lliw  = 10
nst   = 474
nfe   = 610   nfel  = 649
nmi   = 607   nli   = 649
nsetups = 78   netf  = 27
npe   = 8     nps   = 1204
ncfn  = 0     ncfl  = 0
```

### 3 Parallel example problems

#### 3.1 A nonstiff example: pvnx

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

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0.5 \frac{\partial u}{\partial x} \quad (7)$$

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

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

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

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

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

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  $2 \times 2$  blocks, and the corresponding diagonal blocks of the Jacobian are saved each time the preconditioner is generated, for re-use later under certain conditions.

The organization of the 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  $2 \times 2$  array) with a  $5 \times 5$  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 = 306 order = 4 stepsize = 2.06e+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 = 347 order = 4 stepsize = 6.76e+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 = 405 order = 4 stepsize = 3.31e+02  
At bottom left: c1, c2 = 3.497e-08 3.382e+11  
At top right: c1, c2 = 3.674e-07 3.804e+11

t = 5.04e+04 no. steps = 419 order = 5 stepsize = 3.76e+02  
At bottom left: c1, c2 = 6.654e-11 3.358e+11  
At top right: c1, c2 = 3.781e-10 3.864e+11

t = 5.76e+04 no. steps = 432 order = 5 stepsize = 3.80e+02  
At bottom left: c1, c2 = -7.307e-11 3.320e+11  
At top right: c1, c2 = -4.036e-10 3.909e+11

t = 6.48e+04 no. steps = 446 order = 5 stepsize = 6.93e+02  
At bottom left: c1, c2 = -5.582e-10 3.313e+11  
At top right: c1, c2 = -3.105e-09 3.963e+11

t = 7.20e+04 no. steps = 457 order = 5 stepsize = 6.93e+02  
At bottom left: c1, c2 = -2.172e-11 3.330e+11  
At top right: c1, c2 = -1.205e-10 4.039e+11

t = 7.92e+04 no. steps = 467 order = 5 stepsize = 6.93e+02  
At bottom left: c1, c2 = 2.011e-12 3.334e+11  
At top right: c1, c2 = 1.118e-11 4.120e+11

t = 8.64e+04 no. steps = 478 order = 5 stepsize = 6.93e+02  
At bottom left: c1, c2 = 1.871e-15 3.352e+11  
At top right: c1, c2 = 1.007e-14 4.163e+11

Final Statistics:

lenrw	=	2000	leniw	=	80
llrw	=	2046	lliw	=	80
nst	=	478			
nfe	=	611	nfel	=	650
nni	=	608	nli	=	650
nsetups	=	80	netf	=	28
npe	=	9	nps	=	1203
ncfn	=	0	ncfl	=	1

### 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  $5 \times 5$  subgrids on a  $2 \times 2$  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 = 3.00e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

t = 2.88e+04  no. steps = 290  order = 3  stepsize = 1.52e+02
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11
```

```

t = 3.60e+04  no. steps = 342  order = 4  stepsize = 9.02e+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 = 404  order = 4  stepsize = 5.15e+02
At bottom left:  c1, c2 = -1.454e-07  3.382e+11
At top right:    c1, c2 = -1.611e-07  3.804e+11

t = 5.04e+04  no. steps = 420  order = 4  stepsize = 3.57e+02
At bottom left:  c1, c2 = 5.214e-11  3.358e+11
At top right:    c1, c2 = -1.638e-11  3.864e+11

t = 5.76e+04  no. steps = 433  order = 5  stepsize = 3.98e+02
At bottom left:  c1, c2 = -1.024e-11  3.320e+11
At top right:    c1, c2 = 2.802e-10  3.909e+11

t = 6.48e+04  no. steps = 442  order = 5  stepsize = 8.23e+02
At bottom left:  c1, c2 = 2.478e-09  3.313e+11
At top right:    c1, c2 = 3.680e-10  3.963e+11

t = 7.20e+04  no. steps = 451  order = 5  stepsize = 8.23e+02
At bottom left:  c1, c2 = -3.825e-09  3.330e+11
At top right:    c1, c2 = -2.335e-10  4.039e+11

t = 7.92e+04  no. steps = 459  order = 5  stepsize = 8.23e+02
At bottom left:  c1, c2 = -3.604e-11  3.334e+11
At top right:    c1, c2 = 2.031e-11  4.120e+11

t = 8.64e+04  no. steps = 468  order = 5  stepsize = 8.23e+02
At bottom left:  c1, c2 = -4.944e-13  3.352e+11
At top right:    c1, c2 = 1.870e-12  4.162e+11

```

Final Statistics:

```

lenrw  = 2000    leniw = 80
llrw   = 2046    lliw  = 80
nst    = 468
nfe    = 623    nfel  = 593
nni    = 620    nli   = 593
nsetups = 88    netf  = 34
npe    = 9      nps   = 1156
ncfn   = 0      ncfl  = 0

```

```

In CVBBDPRE: real/integer local work space sizes = 600, 50
              no. floccal 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 = 306  order = 3  stepsize = 2.00e+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 = 350  order = 4  stepsize = 7.54e+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 = 410  order = 4  stepsize = 4.51e+02
At bottom left:  c1, c2 = -3.252e-09  3.382e+11
At top right:    c1, c2 = -3.998e-09  3.804e+11

t = 5.04e+04  no. steps = 427  order = 5  stepsize = 4.57e+02
At bottom left:  c1, c2 = 1.089e-11  3.358e+11
At top right:    c1, c2 = 7.119e-12  3.864e+11

t = 5.76e+04  no. steps = 446  order = 3  stepsize = 2.05e+02
At bottom left:  c1, c2 = 3.201e-11  3.320e+11
At top right:    c1, c2 = -7.813e-13  3.909e+11

t = 6.48e+04  no. steps = 463  order = 5  stepsize = 5.79e+02
At bottom left:  c1, c2 = -1.682e-15  3.313e+11
At top right:    c1, c2 = -8.607e-16  3.963e+11

t = 7.20e+04  no. steps = 476  order = 5  stepsize = 5.79e+02
At bottom left:  c1, c2 = -2.820e-16  3.330e+11
At top right:    c1, c2 = -7.249e-18  4.039e+11

t = 7.92e+04  no. steps = 488  order = 5  stepsize = 5.79e+02
At bottom left:  c1, c2 = -9.170e-18  3.334e+11
At top right:    c1, c2 = -3.133e-19  4.120e+11

t = 8.64e+04  no. steps = 501  order = 5  stepsize = 5.79e+02
At bottom left:  c1, c2 = 9.672e-17  3.352e+11
At top right:    c1, c2 = 4.113e-21  4.163e+11

```

Final Statistics:

```

lenrw = 2000  leniw = 80
llrw  = 2046  lliw  = 80
nst   = 501

```

```
nfe      = 641      nfel = 821
nni      = 638      nli  = 821
nsetups  = 102      netf = 32
npe      = 9        nps  = 1352
ncfn     = 0        ncfl = 0
```

```
In CVBBDPRE: real/integer local work space sizes = 600, 50
              no. floal 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 straightforward implementation of the discretized form of Eqns. (4). In `FCVPSET`, the block-diagonal part of the Jacobian,  $J_{bd}$ , is computed (and copied to `P`) if `JOK = 0`, but is simply copied from `BD` to `P` if `JOK = 1`. In both cases, the preconditioner matrix  $P$  is formed from  $J_{bd}$  and its  $2 \times 2$  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  $10 \times 10$  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 = 0.199394E+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 = 336 order = 5 stepsize = 0.112181E+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 = 389 order = 4 stepsize = 0.428799E+03
c1 (bot.left/middle/top rt.) = 0.162296E-07 0.195126E-04 0.100603E-06
c2 (bot.left/middle/top rt.) = 0.338233E+12 0.135488E+12 0.380352E+12

t = 0.504E+05 no. steps = 410 order = 4 stepsize = 0.407135E+03
c1 (bot.left/middle/top rt.) = -0.176496E-07 -0.106959E-04 -0.380790E-08
c2 (bot.left/middle/top rt.) = 0.335816E+12 0.493028E+12 0.386445E+12

t = 0.576E+05 no. steps = 426 order = 5 stepsize = 0.192012E+03
c1 (bot.left/middle/top rt.) = 0.303262E-09 0.183370E-06 0.673644E-10
c2 (bot.left/middle/top rt.) = 0.332031E+12 0.964982E+12 0.390900E+12

t = 0.648E+05 no. steps = 444 order = 5 stepsize = 0.777577E+03
c1 (bot.left/middle/top rt.) = -0.654307E-10 -0.394025E-07 -0.153374E-10
c2 (bot.left/middle/top rt.) = 0.331303E+12 0.892176E+12 0.396342E+12

t = 0.720E+05 no. steps = 453 order = 5 stepsize = 0.777577E+03
c1 (bot.left/middle/top rt.) = 0.120278E-10 0.725732E-08 0.272181E-11
c2 (bot.left/middle/top rt.) = 0.332972E+12 0.618620E+12 0.403885E+12

t = 0.792E+05 no. steps = 462 order = 5 stepsize = 0.777577E+03
c1 (bot.left/middle/top rt.) = 0.204632E-11 0.123056E-08 0.490941E-12
c2 (bot.left/middle/top rt.) = 0.333441E+12 0.666890E+12 0.412026E+12

t = 0.864E+05 no. steps = 471 order = 5 stepsize = 0.777577E+03
c1 (bot.left/middle/top rt.) = -0.653325E-13 -0.393660E-10 -0.151265E-13
c2 (bot.left/middle/top rt.) = 0.335178E+12 0.910691E+12 0.416250E+12

```

Final statistics:

```

number of steps          = 471      number of f evals.      = 613
number of prec. setups   = 81
number of prec. evals.   = 9        number of prec. solves  = 1187
number of nonl. iters.   = 610      number of lin. iters.   = 636
average Krylov subspace dimension (NLI/NNI) = 0.104262E+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

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

on the time interval  $0 \leq t \leq 1$ . In this case, we use  $\alpha = 10$  and  $N = 10 \cdot \text{NPES}$ , 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 `FCVBBDLOPT`. 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:

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

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 horizontal advection and diffusion in the problem, demands a fairly fine spatial mesh to achieve acceptable resolution.

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

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

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, `max1`, 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 CVMODE 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 CVMODE, 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 represent 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  $NPEX = 8, 16, \text{ and } 32$ , with  $MXSUB = 100, 50, \text{ and } 25$ , 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

```
1  /*
2  * -----
3  * $Revision: 1.19.2.5 $
4  * $Date: 2005/04/14 21:36:39 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
7  *               Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem, with the coding
12 * needed for its solution by CVODE. The problem is from
13 * chemical kinetics, and consists of the following three rate
14 * equations:
15 *   dy1/dt = -.04*y1 + 1.e4*y2*y3
16 *   dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*(y2)^2
17 *   dy3/dt = 3.e7*(y2)^2
18 * on the interval from t = 0.0 to t = 4.e10, with initial
19 * conditions: y1 = 1.0, y2 = y3 = 0. The problem is stiff.
20 * While integrating the system, we also use the rootfinding
21 * feature to find the points at which y1 = 1e-4 or at which
22 * y3 = 0.01. This program solves the problem with the BDF method,
23 * Newton iteration with the CVDENSE dense linear solver, and a
24 * user-supplied Jacobian routine.
25 * It uses a scalar relative tolerance and a vector absolute
26 * tolerance. Output is printed in decades from t = .4 to t = 4.e10.
27 * Run statistics (optional outputs) are printed at the end.
28 * -----
29 */
30
31 #include <stdio.h>
32
33 /* Header files with a description of contents used in cvdx.c */
34
35 #include "sundialstypes.h" /* definition of type reatype */
36 #include "cvode.h"        /* prototypes for CVode* functions and */
37                          /* constants CV_BDF, CV_NEWTON, CV_SV, */
38                          /* CV_NORMAL, CV_SUCCESS, and CV_ROOT_RETURN */
39 #include "cvdense.h"      /* prototype for CVDense */
40 #include "nvector_serial.h" /* definitions of type N_Vector, macro */
41                          /* NV_Ith_S, and prototypes for N_VNew_Serial */
42                          /* and N_VDestroy */
43 #include "dense.h"        /* definition of type DenseMat and macro */
44                          /* DENSE_ELEM */
45
46 /* User-defined vector and matrix accessor macros: Ith, IJth */
47
48 /* These macros are defined in order to write code which exactly matches
49 the mathematical problem description given above.
50
51 Ith(v,i) references the ith component of the vector v, where i is in
52 the range [1..NEQ] and NEQ is defined below. The Ith macro is defined
```

```

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

```

```

107 * Main Program
108 *-----
109 */
110
111 int main()
112 {
113     realtype reltol, t, tout;
114     N_Vector y, abstol;
115     void *cnode_mem;
116     int flag, flagr, iout;
117     int rootsfound[2];
118
119     y = abstol = NULL;
120     cnode_mem = NULL;
121
122     /* Create serial vector of length NEQ for I.C. and abstol */
123     y = N_VNew_Serial(NEQ);
124     if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
125     abstol = N_VNew_Serial(NEQ);
126     if (check_flag((void *)abstol, "N_VNew_Serial", 0)) return(1);
127
128     /* Initialize y */
129     Ith(y,1) = Y1;
130     Ith(y,2) = Y2;
131     Ith(y,3) = Y3;
132
133     /* Set the scalar relative tolerance */
134     reltol = RTOL;
135     /* Set the vector absolute tolerance */
136     Ith(abstol,1) = ATOL1;
137     Ith(abstol,2) = ATOL2;
138     Ith(abstol,3) = ATOL3;
139
140     /*
141      Call CVodeCreate to create the solver memory:
142
143      CV_BDF      specifies the Backward Differentiation Formula
144      CV_NEWTON   specifies a Newton iteration
145
146      A pointer to the integrator problem memory is returned and stored in cnode_mem.
147     */
148
149     cnode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
150     if (check_flag((void *)cnode_mem, "CVodeCreate", 0)) return(1);
151
152     /*
153      Call CVodeMalloc to initialize the integrator memory:
154
155      cnode_mem is the pointer to the integrator memory returned by CVodeCreate
156      f          is the user's right hand side function in y'=f(t,y)
157      T0         is the initial time
158      y          is the initial dependent variable vector
159      CV_SV      specifies scalar relative and vector absolute tolerances
160      &reltol    is a pointer to the scalar relative tolerance

```

```

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

```

```

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

```

```

269 }
270
271 /*
272 *-----
273 * Private helper functions
274 *-----
275 */
276
277 static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3)
278 {
279 #if defined(SUNDIALS_EXTENDED_PRECISION)
280     printf("At t = %0.4Le      y =%14.6Le %14.6Le %14.6Le\n", t, y1, y2, y3);
281 #elif defined(SUNDIALS_DOUBLE_PRECISION)
282     printf("At t = %0.4le      y =%14.6le %14.6le %14.6le\n", t, y1, y2, y3);
283 #else
284     printf("At t = %0.4e      y =%14.6e %14.6e %14.6e\n", t, y1, y2, y3);
285 #endif
286
287     return;
288 }
289
290 static void PrintRootInfo(int root_f1, int root_f2)
291 {
292     printf("    rootsfound[] = %3d %3d\n", root_f1, root_f2);
293
294     return;
295 }
296
297 /*
298 * Get and print some final statistics
299 */
300
301 static void PrintFinalStats(void *cvode_mem)
302 {
303     long int nst, nfe, nsetups, njeD, nfeD, nni, ncfn, netf, nge;
304     int flag;
305
306     flag = CVodeGetNumSteps(cvode_mem, &nst);
307     check_flag(&flag, "CVodeGetNumSteps", 1);
308     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
309     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
310     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
311     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
312     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
313     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
314     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
315     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
316     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
317     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
318
319     flag = CVDenseGetNumJacEvals(cvode_mem, &njeD);
320     check_flag(&flag, "CVDenseGetNumJacEvals", 1);
321     flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeD);
322     check_flag(&flag, "CVDenseGetNumRhsEvals", 1);

```

```

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

```

## B Listing of cvbx.c

```
1  /*
2  * -----
3  * $Revision: 1.17.2.3 $
4  * $Date: 2005/04/06 23:33:41 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem with a banded Jacobian,
12 * with the program for its solution by CVODE.
13 * The problem is the semi-discrete form of the advection-diffusion
14 * equation in 2-D:
15 *   du/dt = d^2 u / dx^2 + .5 du/dx + d^2 u / dy^2
16 * on the rectangle 0 <= x <= 2, 0 <= y <= 1, and the time
17 * interval 0 <= t <= 1. Homogeneous Dirichlet boundary conditions
18 * are posed, and the initial condition is
19 *   u(x,y,t=0) = x(2-x)y(1-y)exp(5xy).
20 * The PDE is discretized on a uniform MX+2 by MY+2 grid with
21 * central differencing, and with boundary values eliminated,
22 * leaving an ODE system of size NEQ = MX*MY.
23 * This program solves the problem with the BDF method, Newton
24 * iteration with the CVBAND band linear solver, and a user-supplied
25 * Jacobian routine.
26 * It uses scalar relative and absolute tolerances.
27 * Output is printed at t = .1, .2, ..., 1.
28 * Run statistics (optional outputs) are printed at the end.
29 * -----
30 */
31
32 #include <stdio.h>
33 #include <stdlib.h>
34 #include <math.h>
35
36 /* Header files with a description of contents used in cvbx.c */
37
38 #include "sundialstypes.h" /* definition of type realtype */
39 #include "cvode.h"        /* prototypes for CVode* functions and constants */
40                          /* CV_BDF, CV_NEWTON, CV_SS, CV_NORMAL, and */
41                          /* CV_SUCCESS */
42 #include "cvband.h"       /* prototype for CVBand */
43 #include "nvector_serial.h" /* definitions of type N_Vector, macro */
44                          /* NV_DATA_S, and prototypes for N_VNew_Serial */
45                          /* and N_VDestroy_Serial */
46 #include "band.h"        /* definitions of type BandMat and macros */
47
48 /* Problem Constants */
49
50 #define XMAX RCONST(2.0) /* domain boundaries */
51 #define YMAX RCONST(1.0)
52 #define MX 10           /* mesh dimensions */
```

```

53 #define MY      5
54 #define NEQ    MX*MY      /* number of equations      */
55 #define ATOL   RCONST(1.0e-5) /* scalar absolute tolerance */
56 #define TO     RCONST(0.0)  /* initial time             */
57 #define T1     RCONST(0.1)  /* first output time        */
58 #define DTOUT  RCONST(0.1)  /* output time increment    */
59 #define NOUT   10          /* number of output times   */
60
61 #define ZERO   RCONST(0.0)
62 #define HALF   RCONST(0.5)
63 #define ONE    RCONST(1.0)
64 #define TWO    RCONST(2.0)
65 #define FIVE   RCONST(5.0)
66
67 /* User-defined vector access macro IJth */
68
69 /* IJth is defined in order to isolate the translation from the
70    mathematical 2-dimensional structure of the dependent variable vector
71    to the underlying 1-dimensional storage.
72    IJth(vdata,i,j) references the element in the vdata array for
73    u at mesh point (i,j), where 1 <= i <= MX, 1 <= j <= MY.
74    The vdata array is obtained via the macro call vdata = NV_DATA_S(v),
75    where v is an N_Vector.
76    The variables are ordered by the y index j, then by the x index i. */
77
78 #define IJth(vdata,i,j) (vdata[(j-1) + (i-1)*MY])
79
80 /* Type : UserData (contains grid constants) */
81
82 typedef struct {
83     realtype dx, dy, hdcoef, hacoef, vdcoef;
84 } *UserData;
85
86 /* Private Helper Functions */
87
88 static void SetIC(N_Vector u, UserData data);
89 static void PrintHeader(realtype reltol, realtype abstol, realtype umax);
90 static void PrintOutput(realtype t, realtype umax, long int nst);
91 static void PrintFinalStats(void *cnode_mem);
92
93 /* Private function to check function return values */
94
95 static int check_flag(void *flagvalue, char *funcname, int opt);
96
97 /* Functions Called by the Solver */
98
99 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
100 static void Jac(long int N, long int mu, long int ml, BandMat J,
101               realtype t, N_Vector u, N_Vector fu, void *jac_data,
102               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
103
104 /*
105  *-----
106  * Main Program

```

```

107  *-----
108  */
109
110  int main(void)
111  {
112      realtype dx, dy, reltol, abstol, t, tout, umax;
113      N_Vector u;
114      UserData data;
115      void *cnode_mem;
116      int iout, flag;
117      long int nst;
118
119      u = NULL;
120      data = NULL;
121      cnode_mem = NULL;
122
123      /* Create a serial vector */
124
125      u = N_VNew_Serial(NEQ); /* Allocate u vector */
126      if(check_flag((void*)u, "N_VNew_Serial", 0)) return(1);
127
128      reltol = ZERO; /* Set the tolerances */
129      abstol = ATOL;
130
131      data = (UserData) malloc(sizeof *data); /* Allocate data memory */
132      if(check_flag((void *)data, "malloc", 2)) return(1);
133      dx = data->dx = XMAX/(MX+1); /* Set grid coefficients in data */
134      dy = data->dy = YMAX/(MY+1);
135      data->hdcoef = ONE/(dx*dx);
136      data->hacoef = HALF/(TWO*dx);
137      data->vdcoef = ONE/(dy*dy);
138
139      SetIC(u, data); /* Initialize u vector */
140
141      /*
142         Call CnodeCreate to create integrator memory
143
144         CV_BDF      specifies the Backward Differentiation Formula
145         CV_NEWTON   specifies a Newton iteration
146
147         A pointer to the integrator problem memory is returned and
148         stored in cnode_mem.
149      */
150
151      cnode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
152      if(check_flag((void *)cnode_mem, "CNodeCreate", 0)) return(1);
153
154      /*
155         Call CNodeMalloc to initialize the integrator memory:
156
157         cnode_mem is the pointer to the integrator memory returned by CNodeCreate
158         f          is the user's right hand side function in  $y'=f(t,y)$ 
159         TO         is the initial time
160         u          is the initial dependent variable vector

```

```

161     CV_SS    specifies scalar relative and absolute tolerances
162     reltol  is the scalar relative tolerance
163     &abstol is a pointer to the scalar absolute tolerance
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, data);
183 if(check_flag(&flag, "CVBandSetJacFn", 1)) return(1);
184
185 /* In loop over output points: call CVode, print results, test for errors */
186
187 umax = N_VMaxNorm(u);
188 PrintHeader(reltol, abstol, umax);
189 for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
190     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
191     if(check_flag(&flag, "CVode", 1)) break;
192     umax = N_VMaxNorm(u);
193     flag = CVodeGetNumSteps(cvode_mem, &nst);
194     check_flag(&flag, "CVodeGetNumSteps", 1);
195     PrintOutput(t, umax, nst);
196 }
197
198 PrintFinalStats(cvode_mem); /* Print some final statistics */
199
200 N_VDestroy_Serial(u); /* Free the u vector */
201 CVodeFree(cvode_mem); /* Free the integrator memory */
202 free(data);          /* Free the user data */
203
204 return(0);
205 }
206
207 /*
208 *-----
209 * Functions called by the solver
210 *-----
211 */
212
213 /* f routine. Compute f(t,u). */
214

```

```

215 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
216 {
217     realtype uij, udn, uup, ult, urt, hordc, horac, verdc, hdiff, hadv, vdiff;
218     realtype *udata, *dudata;
219     int i, j;
220     UserData data;
221
222     udata = NV_DATA_S(u);
223     dudata = NV_DATA_S(udot);
224
225     /* Extract needed constants from data */
226
227     data = (UserData) f_data;
228     hordc = data->hdcoef;
229     horac = data->hacoef;
230     verdc = data->vdcoef;
231
232     /* Loop over all grid points. */
233
234     for (j=1; j <= MY; j++) {
235
236         for (i=1; i <= MX; i++) {
237
238             /* Extract u at x_i, y_j and four neighboring points */
239
240             uij = IJth(udata, i, j);
241             udn = (j == 1) ? ZERO : IJth(udata, i, j-1);
242             uup = (j == MY) ? ZERO : IJth(udata, i, j+1);
243             ult = (i == 1) ? ZERO : IJth(udata, i-1, j);
244             urt = (i == MX) ? ZERO : IJth(udata, i+1, j);
245
246             /* Set diffusion and advection terms and load into udot */
247
248             hdiff = hordc*(ult - TWO*uij + urt);
249             hadv = horac*(urt - ult);
250             vdiff = verdc*(uup - TWO*uij + udn);
251             IJth(dudata, i, j) = hdiff + hadv + vdiff;
252         }
253     }
254 }
255
256 /* Jacobian routine. Compute J(t,u). */
257
258 static void Jac(long int N, long int mu, long int ml, BandMat J,
259               realtype t, N_Vector u, N_Vector fu, void *jac_data,
260               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
261 {
262     long int i, j, k;
263     realtype *kthCol, hordc, horac, verdc;
264     UserData data;
265
266     /*
267     The components of f = udot that depend on u(i,j) are
268     f(i,j), f(i-1,j), f(i+1,j), f(i,j-1), f(i,j+1), with

```

```

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

```

```

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

```

```

377 check_flag(&flag, "CVodeGetNumRhsEvals", 1);
378 flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
379 check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
380 flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
381 check_flag(&flag, "CVodeGetNumErrTestFails", 1);
382 flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
383 check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
384 flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
385 check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
386
387 flag = CVBandGetNumJacEvals(cvode_mem, &njeB);
388 check_flag(&flag, "CVBandGetNumJacEvals", 1);
389 flag = CVBandGetNumRhsEvals(cvode_mem, &nfeB);
390 check_flag(&flag, "CVBandGetNumRhsEvals", 1);
391
392 printf("\nFinal Statistics:\n");
393 printf("nst = %-6ld nfe = %-6ld nsetups = %-6ld nfeB = %-6ld njeB = %ld\n",
394        nst, nfe, nsetups, nfeB, njeB);
395 printf("nni = %-6ld ncfm = %-6ld netf = %ld\n \n",
396        nni, ncfm, netf);
397
398 return;
399 }
400
401 /* Check function return value...
402    opt == 0 means SUNDIALS function allocates memory so check if
403    returned NULL pointer
404    opt == 1 means SUNDIALS function returns a flag so check if
405    flag >= 0
406    opt == 2 means function allocates memory so check if returned
407    NULL pointer */
408
409 static int check_flag(void *flagvalue, char *funcname, int opt)
410 {
411     int *errflag;
412
413     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
414
415     if (opt == 0 && flagvalue == NULL) {
416         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
417                funcname);
418         return(1); }
419
420     /* Check if flag < 0 */
421
422     else if (opt == 1) {
423         errflag = (int *) flagvalue;
424         if (*errflag < 0) {
425             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
426                    funcname, *errflag);
427             return(1); }}
428
429     /* Check if function returned NULL pointer - no memory allocated */
430

```

```
431     else if (opt == 2 && flagvalue == NULL) {
432         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
433             funcname);
434         return(1); }
435
436     return(0);
437 }
```

## C Listing of cvkx.c

```
1  /*
2  * -----
3  * $Revision: 1.17.2.3 $
4  * $Date: 2005/04/06 23:33:41 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * An ODE system is generated from the following 2-species diurnal
12 * kinetics advection-diffusion PDE system in 2 space dimensions:
13 *
14 *  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy$ 
15 *                 $+ Ri(c1,c2,t)$  for  $i = 1,2$ , where
16 *  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$  ,
17 *  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$  ,
18 *  $Kv(y) = Kv0*exp(y/5)$  ,
19 *  $Kh, V, Kv0, q1, q2$ , and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
20 * vary diurnally. The problem is posed on the square
21 *  $0 \leq x \leq 20$ ,  $30 \leq y \leq 50$  (all in km),
22 * with homogeneous Neumann boundary conditions, and for time  $t$  in
23 *  $0 \leq t \leq 86400$  sec (1 day).
24 * The PDE system is treated by central differences on a uniform
25 *  $10 \times 10$  mesh, with simple polynomial initial profiles.
26 * The problem is solved with CVODE, with the BDF/GMRES
27 * method (i.e. using the CVSPGMR linear solver) and the
28 * block-diagonal part of the Newton matrix as a left
29 * preconditioner. A copy of the block-diagonal part of the
30 * Jacobian is saved and conditionally reused within the Precond
31 * routine.
32 * -----
33 */
34
35 #include <stdio.h>
36 #include <stdlib.h>
37 #include <math.h>
38 #include "sundialstypes.h" /* definitions of reatype, TRUE and FALSE */
39 #include "cvode.h" /* CVode* prototypes and various constants */
40 #include "cvspgmr.h" /* prototypes & constants for CVSPGMR solver */
41 #include "smalldense.h" /* use generic DENSE solver in preconditioning */
42 #include "nvector_serial.h" /* definitions of type N_Vector and macro */
43 /* NV_DATA_S */
44 #include "sundialsmath.h" /* contains SQR macro */
45
46 /* Problem Constants */
47
48 #define ZERO RCONST(0.0)
49 #define ONE RCONST(1.0)
50 #define TWO RCONST(2.0)
51
52 #define NUM_SPECIES 2 /* number of species */
```

```

53 #define KH          RCONST(4.0e-6)    /* horizontal diffusivity Kh */
54 #define VEL        RCONST(0.001)     /* advection velocity V      */
55 #define KVO        RCONST(1.0e-8)     /* coefficient in Kv(y)     */
56 #define Q1         RCONST(1.63e-16)   /* coefficients q1, q2, c3  */
57 #define Q2         RCONST(4.66e-16)
58 #define C3         RCONST(3.7e16)
59 #define A3         RCONST(22.62)      /* coefficient in expression for q3(t) */
60 #define A4         RCONST(7.601)      /* coefficient in expression for q4(t) */
61 #define C1_SCALE   RCONST(1.0e6)     /* coefficients in initial profiles */
62 #define C2_SCALE   RCONST(1.0e12)
63
64 #define TO         ZERO                /* initial time */
65 #define NOUT       12                 /* number of output times */
66 #define TWOHR      RCONST(7200.0)     /* number of seconds in two hours */
67 #define HALFDAY    RCONST(4.32e4)     /* number of seconds in a half day */
68 #define PI         RCONST(3.1415926535898) /* pi */
69
70 #define XMIN       ZERO                /* grid boundaries in x */
71 #define XMAX       RCONST(20.0)
72 #define YMIN       RCONST(30.0)       /* grid boundaries in y */
73 #define YMAX       RCONST(50.0)
74 #define XMID       RCONST(10.0)       /* grid midpoints in x,y */
75 #define YMID       RCONST(40.0)
76
77 #define MX         10                 /* MX = number of x mesh points */
78 #define MY         10                 /* MY = number of y mesh points */
79 #define NSMX       20                 /* NSMX = NUM_SPECIES*MX */
80 #define MM         (MX*MY)           /* MM = MX*MY */
81
82 /* CVMalloc Constants */
83
84 #define RTOL       RCONST(1.0e-5)     /* scalar relative tolerance */
85 #define FLOOR      RCONST(100.0)      /* value of C1 or C2 at which tolerances */
86                                     /* change from relative to absolute */
87 #define ATOL       (RTOL*FLOOR)       /* scalar absolute tolerance */
88 #define NEQ        (NUM_SPECIES*MM)   /* NEQ = number of equations */
89
90 /* User-defined vector and matrix accessor macros: IJkth, IJth */
91
92 /* IJkth is defined in order to isolate the translation from the
93 mathematical 3-dimensional structure of the dependent variable vector
94 to the underlying 1-dimensional storage. IJth is defined in order to
95 write code which indexes into small dense matrices with a (row,column)
96 pair, where 1 <= row, column <= NUM_SPECIES.
97
98 IJkth(vdata,i,j,k) references the element in the vdata array for
99 species i at mesh point (j,k), where 1 <= i <= NUM_SPECIES,
100 0 <= j <= MX-1, 0 <= k <= MY-1. The vdata array is obtained via
101 the macro call vdata = NV_DATA_S(v), where v is an N_Vector.
102 For each mesh point (j,k), the elements for species i and i+1 are
103 contiguous within vdata.
104
105 IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
106 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 *cnode_mem, N_Vector u, realtype t);
129 static void PrintFinalStats(void *cnode_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 *cnode_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     is the relative tolerance
195     &abstol   is a pointer to the scalar absolute tolerance      */
196 flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
197 if(check_flag(&flag, "CVodeMalloc", 1)) return(1);
198
199 /* Call CVSpmgr to specify the linear solver CVSPGMR
200     with left preconditioning and the maximum Krylov dimension maxl */
201 flag = CVSpmgr(cvode_mem, PREC_LEFT, 0);
202 if(check_flag(&flag, "CVSpmgr", 1)) return(1);
203
204 /* Set modified Gram-Schmidt orthogonalization, preconditioner
205     setup and solve routines Precond and PSolve, and the pointer
206     to the user-defined block data */
207 flag = CVSpmgrSetGSType(cvode_mem, MODIFIED_GS);
208 if(check_flag(&flag, "CVSpmgrSetGSType", 1)) return(1);
209
210 flag = CVSpmgrSetPreconditioner(cvode_mem, Precond, PSolve, data);
211 if(check_flag(&flag, "CVSpmgrSetPreconditioner", 1)) return(1);
212
213 /* In loop over output points, call CVode, print results, test for error */
214 printf(" \n2-species diurnal advection-diffusion problem\n\n");

```

```

215     for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
216         flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
217         PrintOutput(cvode_mem, u, t);
218         if(check_flag(&flag, "CVode", 1)) break;
219     }
220
221     PrintFinalStats(cvode_mem);
222
223     /* Free memory */
224     N_VDestroy_Serial(u);
225     FreeUserData(data);
226     CVodeFree(cvode_mem);
227
228     return(0);
229 }
230
231 /*
232 *-----
233 * Private helper functions
234 *-----
235 */
236
237 /* Allocate memory for data structure of type UserData */
238
239 static UserData AllocUserData(void)
240 {
241     int jx, jy;
242     UserData data;
243
244     data = (UserData) malloc(sizeof *data);
245
246     for (jx=0; jx < MX; jx++) {
247         for (jy=0; jy < MY; jy++) {
248             (data->P)[jx][jy] = denalloc(NUM_SPECIES);
249             (data->Jbd)[jx][jy] = denalloc(NUM_SPECIES);
250             (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
251         }
252     }
253
254     return(data);
255 }
256
257 /* Load problem constants in data */
258
259 static void InitUserData(UserData data)
260 {
261     data->om = PI/HALFDAY;
262     data->dx = (XMAX-XMIN)/(MX-1);
263     data->dy = (YMAX-YMIN)/(MY-1);
264     data->hdco = KH/SQR(data->dx);
265     data->haco = VEL/(TWO*data->dx);
266     data->vdco = (ONE/SQR(data->dy))*KV0;
267 }
268

```

```

269 /* Free data memory */
270
271 static void FreeUserData(UserData data)
272 {
273     int jx, jy;
274
275     for (jx=0; jx < MX; jx++) {
276         for (jy=0; jy < MY; jy++) {
277             denfree((data->P)[jx][jy]);
278             denfree((data->Jbd)[jx][jy]);
279             denfreepiv((data->pivot)[jx][jy]);
280         }
281     }
282
283     free(data);
284 }
285
286 /* Set initial conditions in u */
287
288 static void SetInitialProfiles(N_Vector u, realtype dx, realtype dy)
289 {
290     int jx, jy;
291     realtype x, y, cx, cy;
292     realtype *udata;
293
294     /* Set pointer to data array in vector u. */
295
296     udata = NV_DATA_S(u);
297
298     /* Load initial profiles of c1 and c2 into u vector */
299
300     for (jy=0; jy < MY; jy++) {
301         y = YMIN + jy*dy;
302         cy = SQR(RCONST(0.1)*(y - YMID));
303         cy = ONE - cy + RCONST(0.5)*SQR(cy);
304         for (jx=0; jx < MX; jx++) {
305             x = XMIN + jx*dx;
306             cx = SQR(RCONST(0.1)*(x - XMID));
307             cx = ONE - cx + RCONST(0.5)*SQR(cx);
308             IJKth(udata,1,jx,jy) = C1_SCALE*cx*cy;
309             IJKth(udata,2,jx,jy) = C2_SCALE*cx*cy;
310         }
311     }
312 }
313
314 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
315
316 static void PrintOutput(void *cnode_mem, N_Vector u, realtype t)
317 {
318     long int nst;
319     int qu, flag;
320     realtype hu, *udata;
321     int mxh = MX/2 - 1, myh = MY/2 - 1, mx1 = MX - 1, my1 = MY - 1;
322

```

```

323     udata = NV_DATA_S(u);
324
325     flag = CVodeGetNumSteps(cvode_mem, &nst);
326     check_flag(&flag, "CVodeGetNumSteps", 1);
327     flag = CVodeGetLastOrder(cvode_mem, &qu);
328     check_flag(&flag, "CVodeGetLastOrder", 1);
329     flag = CVodeGetLastStep(cvode_mem, &hu);
330     check_flag(&flag, "CVodeGetLastStep", 1);
331
332     #if defined(SUNDIALS_EXTENDED_PRECISION)
333     printf("t = %.2Le  no. steps = %ld  order = %d  stepsize = %.2Le\n",
334           t, nst, qu, hu);
335     printf("c1 (bot.left/middle/top rt.) = %12.3Le %12.3Le %12.3Le\n",
336           IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
337     printf("c2 (bot.left/middle/top rt.) = %12.3Le %12.3Le %12.3Le\n",
338           IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
339     #elif defined(SUNDIALS_DOUBLE_PRECISION)
340     printf("t = %.2le  no. steps = %ld  order = %d  stepsize = %.2le\n",
341           t, nst, qu, hu);
342     printf("c1 (bot.left/middle/top rt.) = %12.3le %12.3le %12.3le\n",
343           IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
344     printf("c2 (bot.left/middle/top rt.) = %12.3le %12.3le %12.3le\n",
345           IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
346     #else
347     printf("t = %.2e  no. steps = %ld  order = %d  stepsize = %.2e\n",
348           t, nst, qu, hu);
349     printf("c1 (bot.left/middle/top rt.) = %12.3e %12.3e %12.3e\n",
350           IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
351     printf("c2 (bot.left/middle/top rt.) = %12.3e %12.3e %12.3e\n",
352           IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
353     #endif
354 }
355
356 /* Get and print final statistics */
357
358 static void PrintFinalStats(void *cvode_mem)
359 {
360     long int lenrw, leniw ;
361     long int lenrwSPGMR, leniwSPGMR;
362     long int nst, nfe, nsetups, nni, ncf, netf;
363     long int nli, npe, nps, ncfl, nfeSPGMR;
364     int flag;
365
366     flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
367     check_flag(&flag, "CVodeGetWorkSpace", 1);
368     flag = CVodeGetNumSteps(cvode_mem, &nst);
369     check_flag(&flag, "CVodeGetNumSteps", 1);
370     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
371     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
372     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
373     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
374     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
375     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
376     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);

```

```

377 check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
378 flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
379 check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
380
381 flag = CVSpgmrGetWorkSpace(cvode_mem, &lenrwSPGMR, &leniwSPGMR);
382 check_flag(&flag, "CVSpgmrGetWorkSpace", 1);
383 flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
384 check_flag(&flag, "CVSpgmrGetNumLinIters", 1);
385 flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
386 check_flag(&flag, "CVSpgmrGetNumPrecEvals", 1);
387 flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
388 check_flag(&flag, "CVSpgmrGetNumPrecSolves", 1);
389 flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
390 check_flag(&flag, "CVSpgmrGetNumConvFails", 1);
391 flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);
392 check_flag(&flag, "CVSpgmrGetNumRhsEvals", 1);
393
394 printf("\nFinal Statistics.. \n\n");
395 printf("lenrw   = %5ld   leniw = %5ld\n", lenrw, leniw);
396 printf("llrw    = %5ld   lliw = %5ld\n", lenrwSPGMR, leniwSPGMR);
397 printf("nst     = %5ld\n", nst);
398 printf("nfe     = %5ld   nfe1 = %5ld\n", nfe, nfeSPGMR);
399 printf("nni     = %5ld   nli  = %5ld\n", nni, nli);
400 printf("nsetups = %5ld   netf = %5ld\n", nsetups, netf);
401 printf("npe     = %5ld   nps  = %5ld\n", npe, nps);
402 printf("ncfn    = %5ld   ncfl = %5ld\n", ncfn, ncfl);
403 }
404
405 /* Check function return value...
406     opt == 0 means SUNDIALS function allocates memory so check if
407         returned NULL pointer
408     opt == 1 means SUNDIALS function returns a flag so check if
409         flag >= 0
410     opt == 2 means function allocates memory so check if returned
411         NULL pointer */
412
413 static int check_flag(void *flagvalue, char *funcname, int opt)
414 {
415     int *errflag;
416
417     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
418     if (opt == 0 && flagvalue == NULL) {
419         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
420             funcname);
421         return(1); }
422
423     /* Check if flag < 0 */
424     else if (opt == 1) {
425         errflag = (int *) flagvalue;
426         if (*errflag < 0) {
427             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
428                 funcname, *errflag);
429             return(1); }}
430
431

```

```

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

```

```

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

```

```

539 static int Precond(realtype tn, N_Vector u, N_Vector fu,
540                   booleantype jok, booleantype *jcurPtr, realtype gamma,
541                   void *P_data, N_Vector vtemp1, N_Vector vtemp2,
542                   N_Vector vtemp3)
543 {
544     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
545     realtype **(*P)[MY], **(*Jbd)[MY];
546     long int *(*pivot)[MY], ier;
547     int jx, jy;
548     realtype *udata, **a, **j;
549     UserData data;
550
551     /* Make local copies of pointers in P_data, and of pointer to u's data */
552
553     data = (UserData) P_data;
554     P = data->P;
555     Jbd = data->Jbd;
556     pivot = data->pivot;
557     udata = NV_DATA_S(u);
558
559     if (jok) {
560
561         /* jok = TRUE: Copy Jbd to P */
562
563         for (jy=0; jy < MY; jy++)
564             for (jx=0; jx < MX; jx++)
565                 dencopy(Jbd[jx][jy], P[jx][jy], NUM_SPECIES);
566
567         *jcurPtr = FALSE;
568
569     }
570
571     else {
572         /* jok = FALSE: Generate Jbd from scratch and copy to P */
573
574         /* Make local copies of problem variables, for efficiency. */
575
576         q4coef = data->q4;
577         dely = data->dy;
578         verdco = data->vdco;
579         hordco = data->hdco;
580
581         /* Compute 2x2 diagonal Jacobian blocks (using q4 values
582            computed on the last f call). Load into P. */
583
584         for (jy=0; jy < MY; jy++) {
585             ydn = YMIN + (jy - RCONST(0.5))*dely;
586             yup = ydn + dely;
587             cydn = verdco*exp(RCONST(0.2)*ydn);
588             cyup = verdco*exp(RCONST(0.2)*yup);
589             diag = -(cydn + cyup + TWO*hordco);
590             for (jx=0; jx < MX; jx++) {
591                 c1 = IJKth(udata,1,jx,jy);
592                 c2 = IJKth(udata,2,jx,jy);

```

```

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

```

```
647
648 /* Solve the block-diagonal system Px = r using LU factors stored
649    in P and pivot data in pivot, and return the solution in z. */
650
651 for (jx=0; jx < MX; jx++) {
652     for (jy=0; jy < MY; jy++) {
653         v = &(IJKth(zdata, 1, jx, jy));
654         gesl(P[jx][jy], NUM_SPECIES, pivot[jx][jy], v);
655     }
656 }
657
658 return(0);
659 }
```

## D Listing of pvnx.c

```
1  /*
2  * -----
3  * $Revision: 1.12.2.2 $
4  * $Date: 2005/04/01 21:51:52 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh, George Byrne,
7  *                and Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem, with the program for
12 * its solution by CVODE. The problem is the semi-discrete
13 * form of the advection-diffusion equation in 1-D:
14 *   du/dt = d^2 u / dx^2 + .5 du/dx
15 * on the interval 0 <= x <= 2, and the time interval 0 <= t <= 5.
16 * Homogeneous Dirichlet boundary conditions are posed, and the
17 * initial condition is the following:
18 *   u(x,t=0) = x(2-x)exp(2x) .
19 * The PDE is discretized on a uniform grid of size MX+2 with
20 * central differencing, and with boundary values eliminated,
21 * leaving an ODE system of size NEQ = MX.
22 * This program solves the problem with the option for nonstiff
23 * systems: ADAMS method and functional iteration.
24 * It uses scalar relative and absolute tolerances.
25 * Output is printed at t = .5, 1.0, ..., 5.
26 * Run statistics (optional outputs) are printed at the end.
27 *
28 * This version uses MPI for user routines.
29 * Execute with Number of Processors = N, with 1 <= N <= MX.
30 * -----
31 */
32
33 #include <stdio.h>
34 #include <stdlib.h>
35 #include <math.h>
36 #include "sundialstypes.h" /* definition of reatype */
37 #include "cvsode.h" /* prototypes for CVode* and various constants */
38 #include "nvector_parallel.h" /* definitions of type N_Vector and vector */
39 /* macros, and prototypes for N_Vector */
40 /* functions */
41 #include "mpi.h" /* MPI constants and types */
42
43 /* Problem Constants */
44
45 #define ZERO RCONST(0.0)
46
47 #define XMAX RCONST(2.0) /* domain boundary */
48 #define MX 10 /* mesh dimension */
49 #define NEQ MX /* number of equations */
50 #define ATOL RCONST(1.0e-5) /* scalar absolute tolerance */
51 #define TO ZERO /* initial time */
52 #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 *ckode_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 *ckode_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    ckode_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/((realtyp)(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 ckode_mem.
140 */
141
142 ckode_mem = CVodeCreate(CV_ADAMS, CV_FUNCTIONAL);
143 if(check_flag((void *)ckode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
144
145 flag = CVodeSetFdata(ckode_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     ckode_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    is the relative tolerance
157     &abstol   is a pointer to the scalar absolute tolerance
158 */
159
160 flag = CVodeMalloc(ckode_mem, f, T0, u, CV_SS, reltol, &abstol);

```

```

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

```

```

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

```

```

269 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
270 {
271     realtype ui, ult, urt, hordc, horac, hdiff, hadv;
272     realtype *udata, *dudata, *z;
273     int i;
274     int npes, my_pe, my_length, my_pe_m1, my_pe_p1, last_pe, my_last;
275     UserData data;
276     MPI_Status status;
277     MPI_Comm comm;
278
279     udata = NV_DATA_P(u);
280     dudata = NV_DATA_P(udot);
281
282     /* Extract needed problem constants from data */
283     data = (UserData) f_data;
284     hordc = data->hdcoef;
285     horac = data->hacoef;
286
287     /* Extract parameters for parallel computation. */
288     comm = data->comm;
289     npes = data->npes;          /* Number of processes. */
290     my_pe = data->my_pe;       /* Current process number. */
291     my_length = NV_LOCLENGTH_P(u); /* Number of local elements of u. */
292     z = data->z;
293
294     /* Compute related parameters. */
295     my_pe_m1 = my_pe - 1;
296     my_pe_p1 = my_pe + 1;
297     last_pe = npes - 1;
298     my_last = my_length - 1;
299
300     /* Store local segment of u in the working array z. */
301     for (i = 1; i <= my_length; i++)
302         z[i] = udata[i - 1];
303
304     /* Pass needed data to processes before and after current process. */
305     if (my_pe != 0)
306         MPI_Send(&z[1], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
307     if (my_pe != last_pe)
308         MPI_Send(&z[my_length], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
309
310     /* Receive needed data from processes before and after current process. */
311     if (my_pe != 0)
312         MPI_Recv(&z[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
313     else z[0] = ZERO;
314     if (my_pe != last_pe)
315         MPI_Recv(&z[my_length+1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
316                 &status);
317     else z[my_length + 1] = ZERO;
318
319     /* Loop over all grid points in current process. */
320     for (i=1; i<=my_length; i++) {
321
322         /* Extract u at x_i and two neighboring points */

```

```

323     ui = z[i];
324     ult = z[i-1];
325     urt = z[i+1];
326
327     /* Set diffusion and advection terms and load into udot */
328     hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
329     hadv = horac*(urt - ult);
330     dudata[i-1] = hdiff + hadv;
331 }
332 }
333
334 /* Check function return value...
335     opt == 0 means SUNDIALS function allocates memory so check if
336         returned NULL pointer
337     opt == 1 means SUNDIALS function returns a flag so check if
338         flag >= 0
339     opt == 2 means function allocates memory so check if returned
340         NULL pointer */
341
342 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
343 {
344     int *errflag;
345
346     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
347     if (opt == 0 && flagvalue == NULL) {
348         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
349             id, funcname);
350         return(1); }
351
352     /* Check if flag < 0 */
353     else if (opt == 1) {
354         errflag = (int *) flagvalue;
355         if (*errflag < 0) {
356             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
357                 id, funcname, *errflag);
358             return(1); }}
359
360     /* Check if function returned NULL pointer - no memory allocated */
361     else if (opt == 2 && flagvalue == NULL) {
362         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
363             id, funcname);
364         return(1); }
365
366     return(0);
367 }

```

## E Listing of pvkx.c

```
1 /*
2 * -----
3 * $Revision: 1.14.2.3 $
4 * $Date: 2005/04/06 23:33:48 $
5 * -----
6 * Programmer(s): S. D. Cohen, A. C. Hindmarsh, M. R. Wittman, and
7 *                 Radu Serban @ LLNL
8 * -----
9 * Example problem:
10 *
11 * An ODE system is generated from the following 2-species diurnal
12 * kinetics advection-diffusion PDE system in 2 space dimensions:
13 *
14 *  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy$ 
15 *                 $+ Ri(c1,c2,t)$  for  $i = 1,2$ , where
16 *  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$  ,
17 *  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$  ,
18 *  $Kv(y) = Kv0*exp(y/5)$  ,
19 *  $Kh, V, Kv0, q1, q2,$  and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
20 * vary diurnally. The problem is posed on the square
21 *  $0 \leq x \leq 20,$   $30 \leq y \leq 50$  (all in km),
22 * with homogeneous Neumann boundary conditions, and for time  $t$  in
23 *  $0 \leq t \leq 86400$  sec (1 day).
24 * The PDE system is treated by central differences on a uniform
25 * mesh, with simple polynomial initial profiles.
26 *
27 * The problem is solved by CVODE on NPE processors, treated
28 * as a rectangular process grid of size NPEX by NPEY, with
29 *  $NPE = NPEX*NPEY$ . Each processor contains a subgrid of size MXSUB
30 * by MYSUB of the (x,y) mesh. Thus the actual mesh sizes are
31 *  $MX = MXSUB*NPEX$  and  $MY = MYSUB*NPEY$ , and the ODE system size is
32 *  $neq = 2*MX*MY$ .
33 *
34 * The solution is done with the BDF/GMRES method (i.e. using the
35 * CVSPGMR linear solver) and the block-diagonal part of the
36 * Newton matrix as a left preconditioner. A copy of the
37 * block-diagonal part of the Jacobian is saved and conditionally
38 * reused within the preconditioner routine.
39 *
40 * Performance data and sampled solution values are printed at
41 * selected output times, and all performance counters are printed
42 * on completion.
43 *
44 * This version uses MPI for user routines.
45 *
46 * Execution: mpirun -np N pvkx with  $N = NPEX*NPEY$  (see constants
47 * below).
48 * -----
49 */
50
51 #include <stdio.h>
52 #include <stdlib.h>
```

```

53 #include <math.h>
54 #include "sundialstypes.h" /* definitions of reatype, booleantype, TRUE, */
55                             /* and FALSE */
56 #include "sundialsmath.h" /* definition of macro SQR */
57 #include "cvsode.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 KVO 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 *cnode_mem, int my_pe, MPI_Comm comm,
144     N_Vector u, realtype t);
145 static void PrintFinalStats(void *cnode_mem);
146 static void BSend(MPI_Comm comm,
147     int my_pe, int isubx, int isuby,
148     long int dsize, 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 dsize, long int dsizey,
153     realtype uext[], realtype buffer[]);
154 static void BRecvWait(MPI_Request request[],
155     int isubx, int isuby,
156     long int dsize, 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 = NVAR*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",
213                     npes, NPEX*NPEY);
214         MPI_Finalize();

```

```

215     return(1);
216 }
217
218 /* Set local length */
219 local_N = NVAR*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 ccode_mem.
240 */
241 ccode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
242 if (check_flag((void *)ccode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
243
244 /* Set the pointer to user-defined data */
245 flag = CVodeSetFdata(ccode_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     ccode_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     is the relative tolerance
257     &abstol   is a pointer to the scalar absolute tolerance
258 */
259 flag = CVodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
260 if (check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
261
262 /* Call CVSpGmr to specify the linear solver CVSPGMR
263     with left preconditioning and the maximum Krylov dimension maxl */
264 flag = CVSpGmr(ccode_mem, PREC_LEFT, 0);
265 if (check_flag(&flag, "CVSpGmr", 1, my_pe)) MPI_Abort(comm, 1);
266
267 /* Set preconditioner setup and solve routines Precond and PSolve,
268     and the pointer to the user-defined block data */

```

```

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

```

```

323 static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
324 {
325     int isubx, isuby;
326
327     /* Set problem constants */
328     data->om = PI/HALFDAY;
329     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
330     data->dy = (YMAX-YMIN)/((realtype)(MY-1));
331     data->hdco = KH/SQR(data->dx);
332     data->haco = VEL/(RCONST(2.0)*data->dx);
333     data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;
334
335     /* Set machine-related constants */
336     data->comm = comm;
337     data->my_pe = my_pe;
338
339     /* isubx and isuby are the PE grid indices corresponding to my_pe */
340     isuby = my_pe/NPEX;
341     isubx = my_pe - isuby*NPEX;
342     data->isubx = isubx;
343     data->isuby = isuby;
344
345     /* Set the sizes of a boundary x-line in u and uext */
346     data->nvmxsub = NVAR*MXSUB;
347     data->nvmxsub2 = NVAR*(MXSUB+2);
348 }
349
350 /* Free preconditioner data memory */
351
352 static void FreePreconData(PreconData pdata)
353 {
354     int lx, ly;
355
356     for (lx = 0; lx < MXSUB; lx++) {
357         for (ly = 0; ly < MYSUB; ly++) {
358             denfree((pdata->P)[lx][ly]);
359             denfree((pdata->Jbd)[lx][ly]);
360             denfreepiv((pdata->pivot)[lx][ly]);
361         }
362     }
363
364     free(pdata);
365 }
366
367 /* Set initial conditions in u */
368
369 static void SetInitialProfiles(N_Vector u, UserData data)
370 {
371     int isubx, isuby, lx, ly, jx, jy;
372     long int offset;
373     realtype dx, dy, x, y, cx, cy, xmid, ymid;
374     realtype *udata;
375
376     /* Set pointer to data array in vector u */

```

```

377  udata = NV_DATA_P(u);
378
379  /* Get mesh spacings, and subgrid indices for this PE */
380  dx = data->dx;      dy = data->dy;
381  isubx = data->isubx;  isuby = data->isuby;
382
383  /* Load initial profiles of c1 and c2 into local u vector.
384  Here lx and ly are local mesh point indices on the local subgrid,
385  and jx and jy are the global mesh point indices. */
386  offset = 0;
387  xmid = RCONST(0.5)*(XMIN + XMAX);
388  ymid = RCONST(0.5)*(YMIN + YMAX);
389  for (ly = 0; ly < MYSUB; ly++) {
390      jy = ly + isuby*MYSUB;
391      y = YMIN + jy*dy;
392      cy = SQR(RCONST(0.1)*(y - ymid));
393      cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
394      for (lx = 0; lx < MXSUB; lx++) {
395          jx = lx + isubx*MXSUB;
396          x = XMIN + jx*dx;
397          cx = SQR(RCONST(0.1)*(x - xmid));
398          cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
399          udata[offset ] = C1_SCALE*cx*cy;
400          udata[offset+1] = C2_SCALE*cx*cy;
401          offset = offset + 2;
402      }
403  }
404 }
405
406 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
407
408 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
409                        N_Vector u, realtype t)
410 {
411     int qu, flag;
412     realtype hu, *udata, tempu[2];
413     int npelast;
414     long int i0, i1, nst;
415     MPI_Status status;
416
417     npelast = NPEX*NPEY - 1;
418     udata = NV_DATA_P(u);
419
420     /* Send c1,c2 at top right mesh point to PE 0 */
421     if (my_pe == npelast) {
422         i0 = NVAR*MXSUB*MYSUB - 2;
423         i1 = i0 + 1;
424         if (npelast != 0)
425             MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
426         else {
427             tempu[0] = udata[i0];
428             tempu[1] = udata[i1];
429         }
430     }

```

```

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

```

```

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

```

```

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

```

```

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

```

```

647     }
648 }
649 }
650
651 /* ucomm routine. This routine performs all communication
652    between processors of data needed to calculate f. */
653
654 static void ucomm(realtype t, N_Vector u, UserData data)
655 {
656
657     realtype *udata, *uext, buffer[2*NVARS*MYSUB];
658     MPI_Comm comm;
659     int my_pe, isubx, isuby;
660     long int nvmxsub, nvmysub;
661     MPI_Request request[4];
662
663     udata = NV_DATA_P(u);
664
665     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
666     comm = data->comm; my_pe = data->my_pe;
667     isubx = data->isubx; isuby = data->isuby;
668     nvmxsub = data->nvmxsub;
669     nvmysub = NVARS*MYSUB;
670     uext = data->uext;
671
672     /* Start receiving boundary data from neighboring PEs */
673     BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);
674
675     /* Send data from boundary of local grid to neighboring PEs */
676     BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);
677
678     /* Finish receiving boundary data from neighboring PEs */
679     BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
680 }
681
682 /* fcalc routine. Compute f(t,y). This routine assumes that communication
683    between processors of data needed to calculate f has already been done,
684    and this data is in the work array uext. */
685
686 static void fcalc(realtype t, realtype udata[],
687                  realtype dudata[], UserData data)
688 {
689     realtype *uext;
690     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
691     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
692     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
693     realtype q4coef, dely, verdco, hordco, horaco;
694     int i, lx, ly, jx, jy;
695     int isubx, isuby;
696     long int nvmxsub, nvmxsub2, offsetu, offsetue;
697
698     /* Get subgrid indices, data sizes, extended work array uext */
699     isubx = data->isubx; isuby = data->isuby;
700     nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;

```

```

701  uext = data->uext;
702
703  /* Copy local segment of u vector into the working extended array uext */
704  offsetu = 0;
705  offsetue = nvmxsub2 + NVAR;
706  for (ly = 0; ly < MYSUB; ly++) {
707      for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
708      offsetu = offsetu + nvmxsub;
709      offsetue = offsetue + nvmxsub2;
710  }
711
712  /* To facilitate homogeneous Neumann boundary conditions, when this is
713  a boundary PE, copy data from the first interior mesh line of u to uext */
714
715  /* If isuby = 0, copy x-line 2 of u to uext */
716  if (isuby == 0) {
717      for (i = 0; i < nvmxsub; i++) uext[NVAR+i] = udata[nvmxsub+i];
718  }
719
720  /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
721  if (isuby == NPEY-1) {
722      offsetu = (MYSUB-2)*nvmxsub;
723      offsetue = (MYSUB+1)*nvmxsub2 + NVAR;
724      for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
725  }
726
727  /* If isubx = 0, copy y-line 2 of u to uext */
728  if (isubx == 0) {
729      for (ly = 0; ly < MYSUB; ly++) {
730          offsetu = ly*nvmxsub + NVAR;
731          offsetue = (ly+1)*nvmxsub2;
732          for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
733      }
734  }
735
736  /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
737  if (isubx == NPEX-1) {
738      for (ly = 0; ly < MYSUB; ly++) {
739          offsetu = (ly+1)*nvmxsub - 2*NVAR;
740          offsetue = (ly+2)*nvmxsub2 - NVAR;
741          for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
742      }
743  }
744
745  /* Make local copies of problem variables, for efficiency */
746  dely = data->dy;
747  verdco = data->vdco;
748  hordco = data->hdco;
749  horaco = data->haco;
750
751  /* Set diurnal rate coefficients as functions of t, and save q4 in
752  data block for use by preconditioner evaluation routine */
753  s = sin((data->om)*t);
754  if (s > RCONST(0.0)) {

```

```

755     q3 = exp(-A3/s);
756     q4coef = exp(-A4/s);
757 } else {
758     q3 = RCONST(0.0);
759     q4coef = RCONST(0.0);
760 }
761 data->q4 = q4coef;
762
763 /* Loop over all grid points in local subgrid */
764 for (ly = 0; ly < MYSUB; ly++) {
765
766     jy = ly + isuby*MYSUB;
767
768     /* Set vertical diffusion coefficients at jy +- 1/2 */
769     ydn = YMIN + (jy - RCONST(0.5))*dely;
770     yup = ydn + dely;
771     cydn = verdco*exp(RCONST(0.2)*ydn);
772     cyup = verdco*exp(RCONST(0.2)*yup);
773     for (lx = 0; lx < MXSUB; lx++) {
774
775         jx = lx + isubx*MXSUB;
776
777         /* Extract c1 and c2, and set kinetic rate terms */
778         offsetue = (lx+1)*NVARs + (ly+1)*nvmxsub2;
779         c1 = uext[offsetue];
780         c2 = uext[offsetue+1];
781         qq1 = Q1*c1*C3;
782         qq2 = Q2*c1*c2;
783         qq3 = q3*C3;
784         qq4 = q4coef*c2;
785         rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
786         rkin2 = qq1 - qq2 - qq4;
787
788         /* Set vertical diffusion terms */
789         c1dn = uext[offsetue-nvmxsub2];
790         c2dn = uext[offsetue-nvmxsub2+1];
791         c1up = uext[offsetue+nvmxsub2];
792         c2up = uext[offsetue+nvmxsub2+1];
793         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
794         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
795
796         /* Set horizontal diffusion and advection terms */
797         c1lt = uext[offsetue-2];
798         c2lt = uext[offsetue-1];
799         c1rt = uext[offsetue+2];
800         c2rt = uext[offsetue+3];
801         hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
802         hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
803         horad1 = horaco*(c1rt - c1lt);
804         horad2 = horaco*(c2rt - c2lt);
805
806         /* Load all terms into dudata */
807         offsetu = lx*NVARs + ly*nvmxsub;
808         dudata[offsetu] = vertd1 + hord1 + horad1 + rkin1;

```

```

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

```

```

863 /* jok = TRUE: Copy Jbd to P */
864   for (ly = 0; ly < MYSUB; ly++)
865     for (lx = 0; lx < MXSUB; lx++)
866       dencopy(Jbd[lx][ly], P[lx][ly], NVARs);
867
868 *jcurPtr = FALSE;
869
870 }
871
872 else {
873
874 /* jok = FALSE: Generate Jbd from scratch and copy to P */
875
876 /* Make local copies of problem variables, for efficiency */
877 q4coef = data->q4;
878 dely = data->dy;
879 verdco = data->vdco;
880 hordco = data->hdco;
881
882 /* Compute 2x2 diagonal Jacobian blocks (using q4 values
883    computed on the last f call). Load into P. */
884   for (ly = 0; ly < MYSUB; ly++) {
885     jy = ly + isuby*MYSUB;
886     ydn = YMIN + (jy - RCONST(0.5))*dely;
887     yup = ydn + dely;
888     cydn = verdco*exp(RCONST(0.2)*ydn);
889     cyup = verdco*exp(RCONST(0.2)*yup);
890     diag = -(cydn + cyup + RCONST(2.0)*hordco);
891     for (lx = 0; lx < MXSUB; lx++) {
892       jx = lx + isubx*MXSUB;
893       offset = lx*NVARs + ly*nvmxsub;
894       c1 = udata[offset];
895       c2 = udata[offset+1];
896       j = Jbd[lx][ly];
897       a = P[lx][ly];
898       IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
899       IJth(j,1,2) = -Q2*c1 + q4coef;
900       IJth(j,2,1) = Q1*C3 - Q2*c2;
901       IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
902       dencopy(j, a, NVARs);
903     }
904   }
905
906 *jcurPtr = TRUE;
907
908 }
909
910 /* Scale by -gamma */
911   for (ly = 0; ly < MYSUB; ly++)
912     for (lx = 0; lx < MXSUB; lx++)
913       denscale(-gamma, P[lx][ly], NVARs);
914
915 /* Add identity matrix and do LU decompositions on blocks in place */
916   for (lx = 0; lx < MXSUB; lx++) {

```

```

917     for (ly = 0; ly < MYSUB; ly++) {
918         denaddI(P[lx][ly], NVAR);
919         ier = gefa(P[lx][ly], NVAR, pivot[lx][ly]);
920         if (ier != 0) return(1);
921     }
922 }
923
924 return(0);
925 }
926
927 /* Preconditioner solve routine */
928 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
929                 N_Vector r, N_Vector z,
930                 realtype gamma, realtype delta,
931                 int lr, void *P_data, N_Vector vtemp)
932 {
933     realtype **(*P)[MYSUB];
934     long int nvmxsub, *(*pivot)[MYSUB];
935     int lx, ly;
936     realtype *zdata, *v;
937     PreconData predata;
938     UserData data;
939
940     /* Extract the P and pivot arrays from P_data */
941     predata = (PreconData) P_data;
942     data = (UserData) (predata->f_data);
943     P = predata->P;
944     pivot = predata->pivot;
945
946     /* Solve the block-diagonal system Px = r using LU factors stored
947        in P and pivot data in pivot, and return the solution in z.
948        First copy vector r to z. */
949     N_VScale(RCONST(1.0), r, z);
950
951     nvmxsub = data->nvmxsub;
952     zdata = NV_DATA_P(z);
953
954     for (lx = 0; lx < MXSUB; lx++) {
955         for (ly = 0; ly < MYSUB; ly++) {
956             v = &(zdata[lx*NVAR + ly*nvmxsub]);
957             gesl(P[lx][ly], NVAR, pivot[lx][ly], v);
958         }
959     }
960
961     return(0);
962 }
963
964
965 /***** Private Helper Function *****/
966
967 /* Check function return value...
968    opt == 0 means SUNDIALS function allocates memory so check if
969        returned NULL pointer
970    opt == 1 means SUNDIALS function returns a flag so check if

```

```

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

```

## F Listing of pvkxb.c

```
1  /*
2  * -----
3  * $Revision: 1.19.2.2 $
4  * $Date: 2005/04/01 21:51:52 $
5  * -----
6  * Programmer(s): S. D. Cohen, A. C. Hindmarsh, M. R. Wittman, and
7  *                 Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * An ODE system is generated from the following 2-species diurnal
12 * kinetics advection-diffusion PDE system in 2 space dimensions:
13 *
14 *  $dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy$ 
15 *                 $+ Ri(c1,c2,t)$  for  $i = 1,2$ , where
16 *  $R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2$  ,
17 *  $R2(c1,c2,t) = q1*c1*c3 - q2*c1*c2 - q4(t)*c2$  ,
18 *  $Kv(y) = Kv0*exp(y/5)$  ,
19 *  $Kh, V, Kv0, q1, q2$ , and  $c3$  are constants, and  $q3(t)$  and  $q4(t)$ 
20 * vary diurnally. The problem is posed on the square
21 *  $0 \leq x \leq 20$ ,  $30 \leq y \leq 50$  (all in km),
22 * with homogeneous Neumann boundary conditions, and for time  $t$  in
23 *  $0 \leq t \leq 86400$  sec (1 day).
24 * The PDE system is treated by central differences on a uniform
25 * mesh, with simple polynomial initial profiles.
26 *
27 * The problem is solved by CVODE on NPE processors, treated
28 * as a rectangular process grid of size NPEX by NPEY, with
29 *  $NPE = NPEX*NPEY$ . Each processor contains a subgrid of size MXSUB
30 * by MYSUB of the (x,y) mesh. Thus the actual mesh sizes are
31 *  $MX = MXSUB*NPEX$  and  $MY = MYSUB*NPEY$ , and the ODE system size is
32 *  $neq = 2*MX*MY$ .
33 *
34 * The solution is done with the BDF/GMRES method (i.e. using the
35 * CVSPGMR linear solver) and a block-diagonal matrix with banded
36 * blocks as a preconditioner, using the CVBBDPRE module.
37 * Each block is generated using difference quotients, with
38 * half-bandwidths  $mudq = mldq = 2*MXSUB$ , but the retained banded
39 * blocks have half-bandwidths  $mukeep = mlkeep = 2$ .
40 * A copy of the approximate Jacobian is saved and conditionally
41 * reused within the preconditioner routine.
42 *
43 * The problem is solved twice -- with left and right preconditioning.
44 *
45 * Performance data and sampled solution values are printed at
46 * selected output times, and all performance counters are printed
47 * on completion.
48 *
49 * This version uses MPI for user routines.
50 * Execute with number of processors =  $NPEX*NPEY$  (see constants below).
51 * -----
52 */
```

```

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 NVAR 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 KVO 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 */

```

```

107                                     /* change from relative to absolute      */
108 #define ATOL      (RTOL*FLOOR)      /* scalar absolute tolerance */
109
110 /* Type : UserData
111     contains problem constants, extended dependent variable array,
112     grid constants, processor indices, MPI communicator */
113
114 typedef struct {
115     realtype q4, om, dx, dy, hdco, haco, vdco;
116     realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
117     int my_pe, isubx, isuby;
118     long int nvmxsub, nvmxsub2, Nlocal;
119     MPI_Comm comm;
120 } *UserData;
121
122 /* Prototypes of private helper functions */
123
124 static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
125                          UserData data);
126 static void SetInitialProfiles(N_Vector u, UserData data);
127 static void PrintIntro(int npes, long int mudq, long int mldq,
128                       long int mukeep, long int mlkeep);
129 static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
130                        N_Vector u, realtype t);
131 static void PrintFinalStats(void *cvode_mem, void *pdata);
132 static void BSend(MPI_Comm comm,
133                  int my_pe, int isubx, int isuby,
134                  long int dsize, long int dsizey,
135                  realtype uarray[]);
136 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
137                      int my_pe, int isubx, int isuby,
138                      long int dsize, long int dsizey,
139                      realtype uext[], realtype buffer[]);
140 static void BRecvWait(MPI_Request request[],
141                      int isubx, int isuby,
142                      long int dsize, realtype uext[],
143                      realtype buffer[]);
144
145 static void fucomm(realtype t, N_Vector u, void *f_data);
146
147 /* Prototype of function called by the solver */
148
149 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
150
151 /* Prototype of functions called by the CVBBDPRE module */
152
153 static void flocal(long int Nlocal, realtype t, N_Vector u,
154                  N_Vector udot, void *f_data);
155
156 /* Private function to check function return values */
157
158 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
159
160 /***** Main Program *****/

```

```

161
162 int main(int argc, char *argv[])
163 {
164     UserData data;
165     void *cnode_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     cnode_mem = pdata = NULL;
175     u = NULL;
176
177     /* Set problem size neq */
178     neq = NVAR*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 = NVAR*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 is the relative tolerance
234     &abstol is a pointer to the scalar absolute tolerance
235     */
236
237     flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
238     if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
239
240     /* Allocate preconditioner block */
241     mudq = mldq = NVAR*MXSUB;
242     mukeep = mlkeep = NVAR;
243     pdata = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
244                             mukeep, mlkeep, ZERO, flocal, NULL);
245     if(check_flag((void *)pdata, "CVBBDPrecAlloc", 0, my_pe)) MPI_Abort(comm, 1);
246
247     /* Call CVBBDSPgmr to specify the linear solver CVSPGMR using the
248     CVBBDPRE preconditioner, with left preconditioning and the
249     default maximum Krylov dimension maxl */
250     flag = CVBBDSPgmr(cvode_mem, PREC_LEFT, 0, pdata);
251     if(check_flag(&flag, "CVBBDSPgmr", 1, my_pe)) MPI_Abort(comm, 1);
252
253     /* Print heading */
254     if (my_pe == 0) PrintIntro(npes, mudq, mldq, mukeep, mlkeep);
255
256     /* Loop over jpre (= PREC_LEFT, PREC_RIGHT), and solve the problem */
257     for (jpre = PREC_LEFT; jpre <= PREC_RIGHT; jpre++) {
258
259     /* On second run, re-initialize u, the integrator, CVBBDPRE, and CVSPGMR */
260
261     if (jpre == PREC_RIGHT) {
262
263         SetInitialProfiles(u, data);
264
265         flag = CVodeReInit(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
266         if(check_flag(&flag, "CVodeReInit", 1, my_pe)) MPI_Abort(comm, 1);
267
268         flag = CVBBDPrecReInit(pdata, mudq, mldq, ZERO, flocal, NULL);

```

```

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

```

```

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

```

```

377     cx = SQR(RCONST(0.1)*(x - xmid));
378     cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
379     uarray[offset ] = C1_SCALE*cx*cy;
380     uarray[offset+1] = C2_SCALE*cx*cy;
381     offset = offset + 2;
382 }
383 }
384 }
385
386 /* Print problem introduction */
387
388 static void PrintIntro(int npes, long int mudq, long int mldq,
389                      long int mukeep, long int mlkeep)
390 {
391     printf("\n2-species diurnal advection-diffusion problem\n");
392     printf("  %d by %d mesh on %d processors\n", MX, MY, npes);
393     printf("  Using CVBBDPRE preconditioner module\n");
394     printf("  Difference-quotient half-bandwidths are");
395     printf("  mudq = %ld, mldq = %ld\n", mudq, mldq);
396     printf("  Retained band block half-bandwidths are");
397     printf("  mukeep = %ld, mlkeep = %ld", mukeep, mlkeep);
398
399     return;
400 }
401
402 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
403
404 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
405                       N_Vector u, realtype t)
406 {
407     int qu, flag, npelast;
408     long int i0, i1, nst;
409     realtype hu, *uarray, tempu[2];
410     MPI_Status status;
411
412     npelast = NPEX*NPEY - 1;
413     uarray = NV_DATA_P(u);
414
415     /* Send c1,c2 at top right mesh point to PE 0 */
416     if (my_pe == npelast) {
417         i0 = NVAR*MXSUB*MYSUB - 2;
418         i1 = i0 + 1;
419         if (npelast != 0)
420             MPI_Send(&uarray[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
421         else {
422             tempu[0] = uarray[i0];
423             tempu[1] = uarray[i1];
424         }
425     }
426
427     /* On PE 0, receive c1,c2 at top right, then print performance data
428        and sampled solution values */
429     if (my_pe == 0) {
430         if (npelast != 0)

```

```

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

```

```

485     flag = CVSpgmrGetNumLinIters(cvode_mem, &nli);
486     check_flag(&flag, "CVSpgmrGetNumLinIters", 1, 0);
487     flag = CVSpgmrGetNumPrecEvals(cvode_mem, &npe);
488     check_flag(&flag, "CVSpgmrGetNumPrecEvals", 1, 0);
489     flag = CVSpgmrGetNumPrecSolves(cvode_mem, &nps);
490     check_flag(&flag, "CVSpgmrGetNumPrecSolves", 1, 0);
491     flag = CVSpgmrGetNumConvFails(cvode_mem, &ncfl);
492     check_flag(&flag, "CVSpgmrGetNumConvFails", 1, 0);
493     flag = CVSpgmrGetNumRhsEvals(cvode_mem, &nfeSPGMR);
494     check_flag(&flag, "CVSpgmrGetNumRhsEvals", 1, 0);
495
496     printf("\nFinal Statistics: \n\n");
497     printf("lenrw   = %5ld    leniw = %5ld\n", lenrw, leniw);
498     printf("llrw    = %5ld    lliw  = %5ld\n", lenrwSPGMR, leniwSPGMR);
499     printf("nst     = %5ld\n"           , nst);
500     printf("nfe    = %5ld    nfel  = %5ld\n" , nfe, nfeSPGMR);
501     printf("nni    = %5ld    nli   = %5ld\n" , nni, nli);
502     printf("nsetups = %5ld    netf  = %5ld\n" , nsetups, netf);
503     printf("npe    = %5ld    nps   = %5ld\n" , npe, nps);
504     printf("ncfn   = %5ld    ncfl  = %5ld\n\n", ncfn, ncfl);
505
506     flag = CVBBDPrecGetWorkSpace(pdata, &lenrwBBDP, &leniwBBDP);
507     check_flag(&flag, "CVBBDPrecGetWorkSpace", 1, 0);
508     flag = CVBBDPrecGetNumGfnEvals(pdata, &ngevalsBBDP);
509     check_flag(&flag, "CVBBDPrecGetNumGfnEvals", 1, 0);
510     printf("In CVBBDPRE: real/integer local work space sizes = %ld, %ld\n",
511           lenrwBBDP, leniwBBDP);
512     printf("           no. flocal evals. = %ld\n", ngevalsBBDP);
513 }
514
515 /* Routine to send boundary data to neighboring PEs */
516
517 static void BSend(MPI_Comm comm,
518                  int my_pe, int isubx, int isuby,
519                  long int dsizex, long int dsizey,
520                  realtype uarray[])
521 {
522     int i, ly;
523     long int offsetu, offsetbuf;
524     realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
525
526     /* If isuby > 0, send data from bottom x-line of u */
527
528     if (isuby != 0)
529         MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
530
531     /* If isuby < NPEY-1, send data from top x-line of u */
532
533     if (isuby != NPEY-1) {
534         offsetu = (MYSUB-1)*dsizex;
535         MPI_Send(&uarray[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
540 if (isubx != 0) {
541     for (ly = 0; ly < MYSUB; ly++) {
542         offsetbuf = ly*NVAR;
543         offsetu = ly*dsizex;
544         for (i = 0; i < NVAR; i++)
545             bufleft[offsetbuf+i] = uarray[offsetu+i];
546     }
547     MPI_Send(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
548 }
549
550 /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
551
552 if (isubx != NPEX-1) {
553     for (ly = 0; ly < MYSUB; ly++) {
554         offsetbuf = ly*NVAR;
555         offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVAR;
556         for (i = 0; i < NVAR; i++)
557             bufright[offsetbuf+i] = uarray[offsetu+i];
558     }
559     MPI_Send(&bufright[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
560 }
561 }
562
563
564 /* Routine to start receiving boundary data from neighboring PEs.
565 Notes:
566 1) buffer should be able to hold 2*NVAR*MYSUB realtype entries, should be
567 passed to both the BRecvPost and BRecvWait functions, and should not
568 be manipulated between the two calls.
569 2) request should have 4 entries, and should be passed in both calls also. */
570
571 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
572                     int my_pe, int isubx, int isuby,
573                     long int dsizex, long int dsizey,
574                     realtype uext[], realtype buffer[])
575 {
576     long int offsetue;
577     /* Have bufleft and bufright use the same buffer */
578     realtype *bufleft = buffer, *bufright = buffer+NVAR*MYSUB;
579
580     /* If isuby > 0, receive data for bottom x-line of uext */
581     if (isuby != 0)
582         MPI_Irecv(&uext[NVAR], dsizex, PVEC_REAL_MPI_TYPE,
583                 my_pe-NPEX, 0, comm, &request[0]);
584
585     /* If isuby < NPEY-1, receive data for top x-line of uext */
586     if (isuby != NPEY-1) {
587         offsetue = NVAR*(1 + (MYSUB+1)*(MXSUB+2));
588         MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
589                 my_pe+NPEX, 0, comm, &request[1]);
590     }
591
592     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */

```

```

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

```

```

647     if (isubx != NPEX-1) {
648         MPI_Wait(&request[3],&status);
649
650         /* Copy the buffer to uext */
651         for (ly = 0; ly < MYSUB; ly++) {
652             offsetbuf = ly*NVAR;
653             offsetue = (ly+2)*dsizex2 - NVAR;
654             for (i = 0; i < NVAR; i++)
655                 uext[offsetue+i] = bufright[offsetbuf+i];
656         }
657     }
658 }
659
660 /* fucomm routine. This routine performs all inter-processor
661    communication of data in u needed to calculate f.          */
662
663 static void fucomm(realtype t, N_Vector u, void *f_data)
664 {
665     UserData data;
666     realtype *uarray, *uext, buffer[2*NVAR*MYSUB];
667     MPI_Comm comm;
668     int my_pe, isubx, isuby;
669     long int nvmxsub, nvmysub;
670     MPI_Request request[4];
671
672     data = (UserData) f_data;
673     uarray = NV_DATA_P(u);
674
675     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
676
677     comm = data->comm; my_pe = data->my_pe;
678     isubx = data->isubx; isuby = data->isuby;
679     nvmxsub = data->nvmxsub;
680     nvmysub = NVAR*MYSUB;
681     uext = data->uext;
682
683     /* Start receiving boundary data from neighboring PEs */
684
685     BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);
686
687     /* Send data from boundary of local grid to neighboring PEs */
688
689     BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, uarray);
690
691     /* Finish receiving boundary data from neighboring PEs */
692
693     BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);
694 }
695
696 /****** Function called by the solver *****/
697
698 /* f routine. Evaluate f(t,y). First call fucomm to do communication of
699    subgrid boundary data into uext. Then calculate f by a call to flocal. */
700

```

```

701 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data)
702 {
703     UserData data;
704
705     data = (UserData) f_data;
706
707     /* Call fucomm to do inter-processor communication */
708
709     fucomm (t, u, f_data);
710
711     /* Call flocal to calculate all right-hand sides */
712
713     flocal (data->Nlocal, t, u, udot, f_data);
714 }
715
716 /***** Functions called by the CVBBDPRE module *****/
717
718 /* flocal routine. Compute f(t,y). This routine assumes that all
719 inter-processor communication of data needed to calculate f has already
720 been done, and this data is in the work array uext. */
721
722 static void flocal(long int Nlocal, realtype t, N_Vector u,
723                  N_Vector udot, void *f_data)
724 {
725     realtype *uext;
726     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
727     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
728     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
729     realtype q4coef, dely, verdco, hordco, horaco;
730     int i, lx, ly, jx, jy;
731     int isubx, isuby;
732     long int nvmxsub, nvmxsub2, offsetu, offsetue;
733     UserData data;
734     realtype *uarray, *duarray;
735
736     uarray = NV_DATA_P(u);
737     duarray = NV_DATA_P(udot);
738
739     /* Get subgrid indices, array sizes, extended work array uext */
740
741     data = (UserData) f_data;
742     isubx = data->isubx; isuby = data->isuby;
743     nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
744     uext = data->uext;
745
746     /* Copy local segment of u vector into the working extended array uext */
747
748     offsetu = 0;
749     offsetue = nvmxsub2 + NVARs;
750     for (ly = 0; ly < MYSUB; ly++) {
751         for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = uarray[offsetu+i];
752         offsetu = offsetu + nvmxsub;
753         offsetue = offsetue + nvmxsub2;
754     }

```

```

755
756 /* To facilitate homogeneous Neumann boundary conditions, when this is
757 a boundary PE, copy data from the first interior mesh line of u to uext */
758
759 /* If isuby = 0, copy x-line 2 of u to uext */
760 if (isuby == 0) {
761     for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = uarray[nvmxsub+i];
762 }
763
764 /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
765 if (isuby == NPEY-1) {
766     offsetu = (MYSUB-2)*nvmxsub;
767     offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
768     for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = uarray[offsetu+i];
769 }
770
771 /* If isubx = 0, copy y-line 2 of u to uext */
772 if (isubx == 0) {
773     for (ly = 0; ly < MYSUB; ly++) {
774         offsetu = ly*nvmxsub + NVARS;
775         offsetue = (ly+1)*nvmxsub2;
776         for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
777     }
778 }
779
780 /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
781 if (isubx == NPEX-1) {
782     for (ly = 0; ly < MYSUB; ly++) {
783         offsetu = (ly+1)*nvmxsub - 2*NVARS;
784         offsetue = (ly+2)*nvmxsub2 - NVARS;
785         for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
786     }
787 }
788
789 /* Make local copies of problem variables, for efficiency */
790
791 dely = data->dy;
792 verdco = data->vdco;
793 hordco = data->hdco;
794 horaco = data->haco;
795
796 /* Set diurnal rate coefficients as functions of t, and save q4 in
797 data block for use by preconditioner evaluation routine */
798
799 s = sin((data->om)*t);
800 if (s > ZERO) {
801     q3 = exp(-A3/s);
802     q4coef = exp(-A4/s);
803 } else {
804     q3 = ZERO;
805     q4coef = ZERO;
806 }
807 data->q4 = q4coef;
808

```

```

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

```

```

863     }
864 }
865 }
866
867 /* Check function return value...
868     opt == 0 means SUNDIALS function allocates memory so check if
869         returned NULL pointer
870     opt == 1 means SUNDIALS function returns a flag so check if
871         flag >= 0
872     opt == 2 means function allocates memory so check if returned
873         NULL pointer */
874
875 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
876 {
877     int *errflag;
878
879     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
880     if (opt == 0 && flagvalue == NULL) {
881         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
882             id, funcname);
883         return(1); }
884
885     /* Check if flag < 0 */
886     else if (opt == 1) {
887         errflag = (int *) flagvalue;
888         if (*errflag < 0) {
889             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
890                 id, funcname, *errflag);
891             return(1); }}
892
893     /* Check if function returned NULL pointer - no memory allocated */
894     else if (opt == 2 && flagvalue == NULL) {
895         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
896             id, funcname);
897         return(1); }
898
899     return(0);
900 }

```

## G Listing of cvkryf.f

```
1 C -----
2 C $Revision: 1.20.2.1 $
3 C $Date: 2005/04/06 23:33:02 $
4 C -----
5 C FCVODE Example Problem: 2D kinetics-transport, preconditioned 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  $0 \leq x \leq 20, 30 \leq y \leq 50$  (all in km),
21 C with homogeneous Neumann boundary conditions, and for time  $t$ 
22 C in  $0 \leq t \leq 86400$  sec (1 day).
23 C The PDE system is treated by central differences on a uniform
24 C  $10 \times 10$  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 DCPY 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 C IMPLICIT NONE
38 C
39 C INTEGER METH, ITMETH, IATOL, INOPT, ITASK, IER, LNCFL, LNPS
40 C INTEGER LNST, LNFE, LNSETUP, LNLI, LNCF, LQ, LH, LNPE, LNLI
41 C INTEGER IOUT, JPRETYPE, IGSTYPE, MAXL
42 C INTEGER*4 IOPT(40)
43 C INTEGER*4 NEQ, MESHX, MESHY, NST, NFE, NPSET, NPE, NPS, NNI
44 C INTEGER*4 NLI, NCFN, NCFL
45 C DOUBLE PRECISION ATOL, AVDIM, T, TOUT, TWOHR, RTOL, FLOOR, DELT
46 C DOUBLE PRECISION U(2,10,10), ROPT(40)
47 C
48 C DATA TWOHR/7200.0D0/, RTOL/1.0D-5/, FLOOR/100.0D0/,
49 C 1 JPRETYPE/1/, IGSTYPE/1/, MAXL/0/, DELT/0.0D0/
50 C DATA LNST/4/, LNFE/5/, LNSETUP/6/, LNLI/7/, LNCF/8/,
51 C 1 LQ/11/, LH/5/, LNPE/18/, LNLI/19/, LNPS/20/, LNCFL/21/
52 C COMMON /PBDIM/ NEQ
```

```

53 C
54 C Set mesh sizes
55     MESHX = 10
56     MESHY = 10
57 C Load Common and initial values in Subroutine INITKX
58     CALL INITKX(MESHX, MESHY, U)
59 C Set other input arguments.
60     NEQ = 2 * MESHX * MESHY
61     T = 0.0D0
62     METH = 2
63     ITMETH = 2
64     IATOL = 1
65     ATOL = RTOL * FLOOR
66     INOPT = 0
67     ITASK = 1
68 C
69     WRITE(6,10) NEQ
70 10  FORMAT('Krylov example problem: '//
71 1    ' Kinetics-transport, NEQ = ', I4/)
72 C
73     CALL FNVINITS(NEQ, IER)
74     IF (IER .NE. 0) THEN
75         WRITE(6,20) IER
76 20  FORMAT('SUNDIALS_ERROR: FNVINITS returned IER = ', I5)
77         STOP
78     ENDIF
79 C
80     CALL FCVMALLOC(T, U, METH, ITMETH, IATOL, RTOL, ATOL,
81 1    INOPT, IOPT, ROPT, IER)
82     IF (IER .NE. 0) THEN
83         WRITE(6,30) IER
84 30  FORMAT('SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
85         CALL FNVFREES
86         STOP
87     ENDIF
88 C
89     CALL FCVSPGMR(JPRETYPE, IGSTYPE, MAXL, DELT, IER)
90     IF (IER .NE. 0) THEN
91         WRITE(6,40) IER
92 40  FORMAT('SUNDIALS_ERROR: FCVSPGMR returned IER = ', I5)
93         CALL FNVFREES
94         CALL FCVFREE
95         STOP
96     ENDIF
97 C
98     CALL FCVSPGMRSETPREC(1, IER)
99 C
100 C Loop over output points, call FCVODE, print sample solution values.
101     TOUT = TWOHR
102     DO 70 IOUT = 1, 12
103 C
104     CALL FCVODE(TOUT, T, U, ITASK, IER)
105 C
106     WRITE(6,50) T, IOPT(LNST), IOPT(LQ), ROPT(LH)

```

```

107 50   FORMAT(/' t = ', E11.3, 5X, 'no. steps = ', I5,
108     1   '   order = ', I3, '   stepsize = ', E14.6)
109     WRITE(6,55) U(1,1,1), U(1,5,5), U(1,10,10),
110     1   U(2,1,1), U(2,5,5), U(2,10,10)
111 55   FORMAT('  c1 (bot.left/middle/top rt.) = ', 3E14.6/
112     1   '  c2 (bot.left/middle/top rt.) = ', 3E14.6)
113 C
114     IF (IER .NE. 0) THEN
115         WRITE(6,60) IER, IOPT(26)
116 60   FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
117     1   '   Linear Solver returned IER = ', I5)
118         CALL FNVFREES
119         CALL FCVFREE
120         STOP
121         ENDIF
122 C
123     TOUT = TOUT + TWOHR
124 70   CONTINUE
125
126 C Print final statistics.
127     NST = IOPT(LNST)
128     NFE = IOPT(LNFE)
129     NPSET = IOPT(LNSETUP)
130     NPE = IOPT(LNPE)
131     NPS = IOPT(LNPS)
132     NNI = IOPT(LNNI)
133     NLI = IOPT(LNLI)
134     AVDIM = DBLE(NLI) / DBLE(NNI)
135     NCFN = IOPT(LNCF)
136     NCFL = IOPT(LNCFL)
137     WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
138     1   NCFL
139 80   FORMAT(///'Final statistics: '//
140     1   ' number of steps          = ', I5, 5X,
141     2   ' number of f evals.       = ', I5/
142     3   ' number of prec. setups = ', I5/
143     4   ' number of prec. evals. = ', I5, 5X,
144     5   ' number of prec. solves = ', I5/
145     6   ' number of nonl. iters. = ', I5, 5X,
146     7   ' number of lin. iters. = ', I5/
147     8   ' average Krylov subspace dimension (NLI/NNI) = ', E14.6/
148     9   ' number of conv. failures.. nonlinear = ', I3,' linear = ', I3)
149 C
150     CALL FCVFREE
151     CALL FNVFREES
152 C
153     STOP
154     END
155
156     SUBROUTINE INITKX(MESHX, MESHY, U0)
157 C Routine to set problem constants and initial values
158 C
159     IMPLICIT NONE
160 C

```

```

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

```

```

215     DOUBLE PRECISION T, U(2,*), UDOT(2,*)
216     DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
217     DOUBLE PRECISION VDCO, HACO
218     DOUBLE PRECISION C1, C2, C1DN, C2DN, C1UP, C2UP, C1LT, C2LT
219     DOUBLE PRECISION C1RT, C2RT, CYDN, CYUP, HORD1, HORD2, HORAD1
220     DOUBLE PRECISION HORAD2, QQ1, QQ2, QQ3, QQ4, RKIN1, RKIN2, S
221     DOUBLE PRECISION VERTD1, VERTD2, YDN, YUP
222 C
223     COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
224     COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
225 C
226 C Set diurnal rate coefficients.
227     S = SIN(OM * T)
228     IF (S .GT. 0.0D0) THEN
229         Q3 = EXP(-A3 / S)
230         Q4 = EXP(-A4 / S)
231     ELSE
232         Q3 = 0.0D0
233         Q4 = 0.0D0
234     ENDIF
235 C
236 C Loop over all grid points.
237     DO 20 JY = 1, MY
238         YDN = 30.0D0 + (JY - 1.5D0) * DY
239         YUP = YDN + DY
240         CYDN = VDCO * EXP(0.2D0 * YDN)
241         CYUP = VDCO * EXP(0.2D0 * YUP)
242         IBLOK0 = (JY - 1) * MX
243         IDN = -MX
244         IF (JY .EQ. 1) IDN = MX
245         IUP = MX
246         IF (JY .EQ. MY) IUP = -MX
247         DO 10 JX = 1, MX
248             IBLOK = IBLOK0 + JX
249             C1 = U(1,IBLOK)
250             C2 = U(2,IBLOK)
251 C Set kinetic rate terms.
252         QQ1 = Q1 * C1 * C3
253         QQ2 = Q2 * C1 * C2
254         QQ3 = Q3 * C3
255         QQ4 = Q4 * C2
256         RKIN1 = -QQ1 - QQ2 + 2.0D0 * QQ3 + QQ4
257         RKIN2 = QQ1 - QQ2 - QQ4
258 C Set vertical diffusion terms.
259         C1DN = U(1,IBLOK + IDN)
260         C2DN = U(2,IBLOK + IDN)
261         C1UP = U(1,IBLOK + IUP)
262         C2UP = U(2,IBLOK + IUP)
263         VERTD1 = CYUP * (C1UP - C1) - CYDN * (C1 - C1DN)
264         VERTD2 = CYUP * (C2UP - C2) - CYDN * (C2 - C2DN)
265 C Set horizontal diffusion and advection terms.
266         ILEFT = -1
267         IF (JX .EQ. 1) ILEFT = 1
268         IRIGHT = 1

```

```

269         IF (JX .EQ. MX) IRIGHT = -1
270         C1LT = U(1,IBLOK + ILEFT)
271         C2LT = U(2,IBLOK + ILEFT)
272         C1RT = U(1,IBLOK + IRIGHT)
273         C2RT = U(2,IBLOK + IRIGHT)
274         HORD1 = HDCO * (C1RT - 2.0D0 * C1 + C1LT)
275         HORD2 = HDCO * (C2RT - 2.0D0 * C2 + C2LT)
276         HORAD1 = HACO * (C1RT - C1LT)
277         HORAD2 = HACO * (C2RT - C2LT)
278     C Load all terms into UDOT.
279         UDOT(1,IBLOK) = VERTD1 + HORD1 + HORAD1 + RKIN1
280         UDOT(2,IBLOK) = VERTD2 + HORD2 + HORAD2 + RKIN2
281     10     CONTINUE
282     20     CONTINUE
283     RETURN
284     END
285
286     SUBROUTINE FCVPSET(T, U, FU, JOK, JCUR, GAMMA, EWT, H,
287     1         V1, V2, V3, IER)
288     C Routine to set and preprocess block-diagonal preconditioner.
289     C Note: The dimensions in /BDJ/ below assume at most 100 mesh points.
290     C
291         IMPLICIT NONE
292     C
293         INTEGER IER, JOK, JCUR, H
294         INTEGER*4 LENBD, JY, JX, IBLOK, MX, MY, MM
295         INTEGER*4 IBLOKO, IPP
296         DOUBLE PRECISION T, U(2,*), GAMMA
297         DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
298         DOUBLE PRECISION VDCO, HACO
299         DOUBLE PRECISION BD, P, FU, EWT, V1, V2, V3
300         DOUBLE PRECISION C1, C2, CYDN, CYUP, DIAG, TEMP, YDN, YUP
301     C
302         COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
303         COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
304         COMMON /BDJ/ BD(2,2,100), P(2,2,100), IPP(2,100)
305     C
306         IER = 0
307         LENBD = 4 * MM
308     C
309     C If JOK = 1, copy BD to P.
310         IF (JOK .EQ. 1) THEN
311             CALL DCOPY(LENBD, BD(1,1,1), 1, P(1,1,1), 1)
312             JCUR = 0
313         ELSE
314     C
315     C JOK = 0. Compute diagonal Jacobian blocks and copy to P.
316     C (using q4 value computed on last FCVFUN call).
317         DO 20 JY = 1, MY
318             YDN = 30.0D0 + (JY - 1.5D0) * DY
319             YUP = YDN + DY
320             CYDN = VDCO * EXP(0.2D0 * YDN)
321             CYUP = VDCO * EXP(0.2D0 * YUP)
322             DIAG = -(CYDN + CYUP + 2.0D0 * HDCO)

```

```

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

```

```

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

```

```

431 c      blas daxpy,dscal,idamax
432 c
433 c      internal variables
434 c
435 c      gaussian elimination with partial pivoting
436 c
437 c      info = 0
438 c      nm1 = n - 1
439 c      if (nm1 .lt. 1) go to 70
440 c      do 60 k = 1, nm1
441 c          kp1 = k + 1
442 c
443 c          find l = pivot index
444 c
445 c          l = idamax(n - k + 1, a(k,k), 1) + k - 1
446 c          ipvt(k) = l
447 c
448 c          zero pivot implies this column already triangularized
449 c
450 c          if (a(l,k) .eq. 0.0d0) go to 40
451 c
452 c          interchange if necessary
453 c
454 c          if (l .eq. k) go to 10
455 c              t = a(l,k)
456 c              a(l,k) = a(k,k)
457 c              a(k,k) = t
458 c      10      continue
459 c
460 c          compute multipliers
461 c
462 c          t = -1.0d0 / a(k,k)
463 c          call dscal(n - k, t, a(k + 1,k), 1)
464 c
465 c          row elimination with column indexing
466 c
467 c          do 30 j = kp1, n
468 c              t = a(l,j)
469 c              if (l .eq. k) go to 20
470 c                  a(l,j) = a(k,j)
471 c                  a(k,j) = t
472 c      20      continue
473 c              call daxpy(n - k, t, a(k + 1,k), 1, a(k + 1,j), 1)
474 c      30      continue
475 c          go to 50
476 c      40      continue
477 c          info = k
478 c      50      continue
479 c      60      continue
480 c      70      continue
481 c          ipvt(n) = n
482 c          if (a(n,n) .eq. 0.0d0) info = n
483 c          return
484 c          end

```

```

485 C
486     subroutine dgesl(a, lda, n, ipvt, b, job)
487 C
488     implicit none
489 C
490     integer lda, n, job, k, kb, l, nm1
491     integer*4 ipvt(1)
492     double precision a(lda,1), b(1), ddot, t
493 C
494 C     dgesl solves the double precision system
495 C     a * x = b  or  trans(a) * x = b
496 C     using the factors computed by dgeco or dgefa.
497 C
498 C     on entry
499 C
500 C         a         double precision(lda, n)
501 C                 the output from dgeco or dgefa.
502 C
503 C         lda       integer
504 C                 the leading dimension of the array  a .
505 C
506 C         n         integer
507 C                 the order of the matrix  a .
508 C
509 C         ipvt      integer(n)
510 C                 the pivot vector from dgeco or dgefa.
511 C
512 C         b         double precision(n)
513 C                 the right hand side vector.
514 C
515 C         job       integer
516 C                 = 0           to solve  a*x = b ,
517 C                 = nonzero    to solve  trans(a)*x = b  where
518 C                             trans(a)  is the transpose.
519 C
520 C     on return
521 C
522 C         b         the solution vector  x .
523 C
524 C     error condition
525 C
526 C         a division by zero will occur if the input factor contains a
527 C         zero on the diagonal.  technically this indicates singularity
528 C         but it is often caused by improper arguments or improper
529 C         setting of lda .  it will not occur if the subroutines are
530 C         called correctly and if dgeco has set rcond .gt. 0.0
531 C         or dgefa has set info .eq. 0 .
532 C
533 C     to compute  inverse(a) * c  where  c  is a matrix
534 C     with  p  columns
535 C         call dgeco(a,lda,n,ipvt,rcond,z)
536 C         if (rcond is too small) go to ...
537 C         do 10 j = 1, p
538 C             call dgesl(a,lda,n,ipvt,c(1,j),0)

```

```

539 c      10 continue
540 c
541 c      linpack. this version dated 08/14/78 .
542 c      cleve moler, university of new mexico, argonne national lab.
543 c
544 c      subroutines and functions
545 c
546 c      blas daxpy,ddot
547 c
548 c      internal variables
549 c
550      nm1 = n - 1
551      if (job .ne. 0) go to 50
552 c
553 c      job = 0 , solve a * x = b
554 c      first solve l*y = b
555 c
556      if (nm1 .lt. 1) go to 30
557      do 20 k = 1, nm1
558          l = ipvt(k)
559          t = b(l)
560          if (l .eq. k) go to 10
561              b(l) = b(k)
562              b(k) = t
563      10      continue
564          call daxpy(n - k, t, a(k + 1,k), 1, b(k + 1), 1)
565      20      continue
566      30      continue
567 c
568 c      now solve u*x = y
569 c
570      do 40 kb = 1, n
571          k = n + 1 - kb
572          b(k) = b(k) / a(k,k)
573          t = -b(k)
574          call daxpy(k - 1, t, a(1,k), 1, b(1), 1)
575      40      continue
576      go to 100
577      50 continue
578 c
579 c      job = nonzero, solve trans(a) * x = b
580 c      first solve trans(u)*y = b
581 c
582      do 60 k = 1, n
583          t = ddot(k - 1, a(1,k), 1, b(1), 1)
584          b(k) = (b(k) - t) / a(k,k)
585      60      continue
586 c
587 c      now solve trans(l)*x = y
588 c
589      if (nm1 .lt. 1) go to 90
590      do 80 kb = 1, nm1
591          k = n - kb
592          b(k) = b(k) + ddot(n - k, a(k + 1,k), 1, b(k + 1), 1)

```

```

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

```

```

647     do 50 i = mp1, n, 4
648         dy(i) = dy(i) + da * dx(i)
649         dy(i + 1) = dy(i + 1) + da * dx(i + 1)
650         dy(i + 2) = dy(i + 2) + da * dx(i + 2)
651         dy(i + 3) = dy(i + 3) + da * dx(i + 3)
652     50 continue
653     return
654     end
655 c
656     subroutine dscal(n, da, dx, incx)
657 c
658 c     scales a vector by a constant.
659 c     uses unrolled loops for increment equal to one.
660 c     jack dongarra, linpack, 3/11/78.
661 c
662     implicit none
663 c
664     integer i, incx, m, mp1, nincx
665     integer*4 n
666     double precision da, dx(1)
667 c
668     if (n.le.0) return
669     if (incx .eq. 1) go to 20
670 c
671 c     code for increment not equal to 1
672 c
673     nincx = n * incx
674     do 10 i = 1, nincx, incx
675         dx(i) = da * dx(i)
676     10 continue
677     return
678 c
679 c     code for increment equal to 1
680 c
681 c
682 c     clean-up loop
683 c
684     20 m = mod(n, 5)
685     if ( m .eq. 0 ) go to 40
686     do 30 i = 1, m
687         dx(i) = da * dx(i)
688     30 continue
689     if ( n .lt. 5 ) return
690     40 mp1 = m + 1
691     do 50 i = mp1, n, 5
692         dx(i) = da * dx(i)
693         dx(i + 1) = da * dx(i + 1)
694         dx(i + 2) = da * dx(i + 2)
695         dx(i + 3) = da * dx(i + 3)
696         dx(i + 4) = da * dx(i + 4)
697     50 continue
698     return
699     end
700 c

```

```

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

```

```

755 c
756 c finds the index of element having max. absolute value.
757 c jack dongarra, linpack, 3/11/78.
758 c
759 c implicit none
760 c
761 c integer i, incx, ix
762 c integer*4 n
763 c double precision dx(1), dmax
764 c
765 c idamax = 0
766 c if (n .lt. 1) return
767 c idamax = 1
768 c if (n .eq. 1) return
769 c if (incx .eq. 1) go to 20
770 c
771 c code for increment not equal to 1
772 c
773 c ix = 1
774 c dmax = abs(dx(1))
775 c ix = ix + incx
776 c do 10 i = 2, n
777 c if (abs(dx(ix)) .le. dmax) go to 5
778 c idamax = i
779 c dmax = abs(dx(ix))
780 c 5 ix = ix + incx
781 c 10 continue
782 c return
783 c
784 c code for increment equal to 1
785 c
786 c 20 dmax = abs(dx(1))
787 c do 30 i = 2, n
788 c if (abs(dx(i)) .le. dmax) go to 30
789 c idamax = i
790 c dmax = abs(dx(i))
791 c 30 continue
792 c return
793 c end
794 c
795 c subroutine dcopy(n, dx, incx, dy, incy)
796 c
797 c copies a vector, x, to a vector, y.
798 c uses unrolled loops for increments equal to one.
799 c jack dongarra, linpack, 3/11/78.
800 c
801 c implicit none
802 c
803 c integer i, incx, incy, ix, iy, m, mp1
804 c integer*4 n
805 c double precision dx(1), dy(1)
806 c
807 c if (n .le. 0) return
808 c if (incx .eq. 1 .and. incy .eq. 1) go to 20

```

```

809 c
810 c      code for unequal increments or equal increments
811 c      not equal to 1
812 c
813     ix = 1
814     iy = 1
815     if (incx .lt. 0) ix = (-n + 1) * incx + 1
816     if (incy .lt. 0) iy = (-n + 1) * incy + 1
817     do 10 i = 1, n
818         dy(iy) = dx(ix)
819         ix = ix + incx
820         iy = iy + incy
821 10 continue
822     return
823 c
824 c      code for both increments equal to 1
825 c
826 c
827 c      clean-up loop
828 c
829     20 m = mod(n, 7)
830     if ( m .eq. 0 ) go to 40
831     do 30 i = 1, m
832         dy(i) = dx(i)
833 30 continue
834     if ( n .lt. 7 ) return
835 40 mp1 = m + 1
836     do 50 i = mp1, n, 7
837         dy(i) = dx(i)
838         dy(i + 1) = dx(i + 1)
839         dy(i + 2) = dx(i + 2)
840         dy(i + 3) = dx(i + 3)
841         dy(i + 4) = dx(i + 4)
842         dy(i + 5) = dx(i + 5)
843         dy(i + 6) = dx(i + 6)
844 50 continue
845     return
846     end

```

## H Listing of pvdiagkbf.f

```
1 C -----
2 C $Revision: 1.18 $
3 C $Date: 2004/10/21 18:58:44 $
4 C -----
5 C Diagonal ODE example. Stiff case, with diagonal preconditioner.
6 C Uses FCVODE interfaces and FCVBBD interfaces.
7 C Solves problem twice -- with left and right preconditioning.
8 C -----
9 C
10 C Include MPI-Fortran header file for MPI_COMM_WORLD, MPI types.
11
12 INCLUDE "mpif.h"
13 C
14 INTEGER NOUT, LNST, LNFE, LNSETUP, LNNI, LNCF, LNETF, LNPE
15 INTEGER LNLI, LNPS, LNCFL, MYPE, IER, NPES, METH, ITMETH
16 INTEGER IATOL, INOPT, ITASK, IPRE, IGS, IOUT
17 INTEGER*4 IOPT(40)
18 INTEGER*4 NEQ, NLOCAL, I, MUDQ, MLDQ, MU, ML, NETF
19 INTEGER*4 NST, NFE, NPSET, NPE, NPS, NNI, NLI, NCFN, NCFL
20 INTEGER*4 LENRPW, LENIPW, NGE
21 DOUBLE PRECISION ALPHA, TOUT, ERMAS, AVDIM
22 DOUBLE PRECISION ATOL, ERRI, RTOL, GERMAX, DTOUT, Y, ROPT, T
23 DIMENSION Y(1024), ROPT(40)
24 C
25 DATA ATOL/1.0D-10/, RTOL/1.0D-5/, DTOUT/0.1D0/, NOUT/10/
26 DATA LNST/4/, LNFE/5/, LNSETUP/6/, LNNI/7/, LNCF/8/, LNETF/9/,
27 1 LNPE/18/, LNLI/19/, LNPS/20/, LNCFL/21/
28 C
29 COMMON /PCOM/ ALPHA, NLOCAL, MYPE
30 C
31 C Get NPES and MYPE. Requires initialization of MPI.
32 CALL MPI_INIT(IER)
33 IF (IER .NE. 0) THEN
34 WRITE(6,5) IER
35 5 FORMAT(///' MPI_ERROR: MPI_INIT returned IER = ', I5)
36 STOP
37 ENDIF
38 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IER)
39 IF (IER .NE. 0) THEN
40 WRITE(6,6) IER
41 6 FORMAT(///' MPI_ERROR: MPI_COMM_SIZE returned IER = ', I5)
42 CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
43 STOP
44 ENDIF
45 CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYPE, IER)
46 IF (IER .NE. 0) THEN
47 WRITE(6,7) IER
48 7 FORMAT(///' MPI_ERROR: MPI_COMM_RANK returned IER = ', I5)
49 CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
50 STOP
51 ENDIF
52
```

```

53  C
54  C   Set input arguments.
55      NLOCAL = 10
56      NEQ = NPES * NLOCAL
57      T = 0.0D0
58      METH = 2
59      ITMETH = 2
60      IATOL = 1
61      INOPT = 0
62      ITASK = 1
63      IPRE = 1
64      IGS = 1
65  C   Set parameter alpha
66      ALPHA = 10.0D0
67  C
68      DO I = 1, NLOCAL
69          Y(I) = 1.0D0
70      ENDDO
71  C
72      IF (MYPE .EQ. 0) THEN
73          WRITE(6,15) NEQ, ALPHA, RTOL, ATOL, NPES
74  15      FORMAT('Diagonal test problem:''' NEQ = ', I3, /
75          &      ' parameter alpha = ', F8.3/
76          &      ' ydot_i = -alpha*i * y_i (i = 1,...,NEQ)'/
77          &      ' RTOL, ATOL = ', 2E10.1/
78          &      ' Method is BDF/NEWTON/SPGMR'/
79          &      ' Preconditioner is band-block-diagonal, using CVBBDPRE'
80          &      '/' Number of processors = ', I3/)
81      ENDIF
82  C
83      CALL FNVINITP(NLOCAL, NEQ, IER)
84  C
85      IF (IER .NE. 0) THEN
86          WRITE(6,20) IER
87  20      FORMAT('///' SUNDIALS_ERROR: FNVINITP returned IER = ', I5)
88          CALL MPI_FINALIZE(IER)
89          STOP
90      ENDIF
91  C
92      CALL FCVMALLOC(T, Y, METH, ITMETH, IATOL, RTOL, ATOL,
93          &      INOPT, IOPT, ROPT, IER)
94  C
95      IF (IER .NE. 0) THEN
96          WRITE(6,30) IER
97  30      FORMAT('///' SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
98          CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
99          STOP
100     ENDIF
101  C
102     MUDQ = 0
103     MLDQ = 0
104     MU = 0
105     ML = 0
106     CALL FCVBBDINIT(NLOCAL, MUDQ, MLDQ, MU, ML, 0.0D0, IER)

```

```

107     IF (IER .NE. 0) THEN
108         WRITE(6,35) IER
109 35     FORMAT(///' SUNDIALS_ERROR: FCVBBDINIT returned IER = ', I5)
110         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
111         STOP
112     ENDIF
113 C
114     CALL FCVBBDSPGMR(IPRE, IGS, 0, 0.0D0, IER)
115     IF (IER .NE. 0) THEN
116         WRITE(6,36) IER
117 36     FORMAT(///' SUNDIALS_ERROR: FCVBBDSPGMR returned IER = ', I5)
118         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
119         STOP
120     ENDIF
121 C
122     IF (MYPE .EQ. 0) WRITE(6,38)
123 38     FORMAT(/'Preconditioning on left'/)
124 C
125 C     Looping point for cases IPRE = 1 and 2.
126 C
127 40     CONTINUE
128 C
129 C     Loop through tout values, call solver, print output, test for failure.
130     TOUT = DTOUT
131     DO 60 IOUT = 1, NOUT
132 C
133         CALL FCVODE(TOUT, T, Y, ITASK, IER)
134 C
135         IF (MYPE .EQ. 0) WRITE(6,45) T, IOPT(LNST), IOPT(LNFE)
136 45     FORMAT(' t = ', E10.2, 5X, 'no. steps = ', I5,
137 &           ' no. f-s = ', I5)
138 C
139         IF (IER .NE. 0) THEN
140             WRITE(6,50) IER, IOPT(26)
141 50     FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
142 &           ' Linear Solver returned IER = ', I5)
143             CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
144             STOP
145         ENDIF
146 C
147         TOUT = TOUT + DTOUT
148 60     CONTINUE
149 C
150 C     Get max. absolute error in the local vector.
151     ERMAX = 0.0D0
152     DO 65 I = 1, NLOCAL
153         ERRI = Y(I) - EXP(-ALPHA * (MYPE * NLOCAL + I) * T)
154         ERMAX = MAX(ERMAX, ABS(ERRI))
155 65     CONTINUE
156 C     Get global max. error from MPI_REDUCE call.
157     CALL MPI_REDUCE(ERMAX, GERMAX, 1, MPI_DOUBLE_PRECISION, MPI_MAX,
158 &                 0, MPI_COMM_WORLD, IER)
159     IF (IER .NE. 0) THEN
160         WRITE(6,70) IER

```

```

161 70      FORMAT(///' MPI_ERROR: MPI_REDUCE returned IER = ', I5)
162      CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
163      STOP
164      ENDIF
165      IF (MYPE .EQ. 0) WRITE(6,75) GERMAX
166 75      FORMAT(/'Max. absolute error is', E10.2/)
167  C
168  C      Print final statistics.
169      IF (MYPE .EQ. 0) THEN
170          NST = IOPT(LNST)
171          NFE = IOPT(LNFE)
172          NPSET = IOPT(LNSETUP)
173          NPE = IOPT(LNPE)
174          NPS = IOPT(LNPS)
175          NNI = IOPT(LNNI)
176          NLI = IOPT(LNLI)
177          AVDIM = DBLE(NLI) / DBLE(NNI)
178          NCFN = IOPT(LNCF)
179          NCFL = IOPT(LNCFL)
180          NETF = IOPT(LNETF)
181          WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
182          &          NCFL, NETF
183 80      FORMAT(/'Final statistics: '//
184          &          ' number of steps          = ', I5, 4X,
185          &          ' number of f evals.          = ', I5/
186          &          ' number of prec. setups = ', I5/
187          &          ' number of prec. evals. = ', I5, 4X,
188          &          ' number of prec. solves = ', I5/
189          &          ' number of nonl. iters. = ', I5, 4X,
190          &          ' number of lin. iters. = ', I5/
191          &          ' average Krylov subspace dimension (NLI/NNI) = ', F8.4/
192          &          ' number of conv. failures.. nonlinear = ', I3,
193          &          ' linear = ', I3/
194          &          ' number of error test failures = ', I3/)
195          CALL FCVBBDOPT(LENRPW, LENIPW, NGE)
196          WRITE(6,82) LENRPW, LENIPW, NGE
197 82      FORMAT('In CVBBDPRE: '//
198          &          ' real/int local workspace = ', 2I5/
199          &          ' number of g evals. = ', I5)
200      ENDIF
201  C
202  C      If IPRE = 1, re-initialize T, Y, and the solver, and loop for case IPRE = 2.
203  C      Otherwise jump to final block.
204      IF (IPRE .EQ. 2) GO TO 99
205  C
206      T = 0.0D0
207      DO I = 1, NLOCAL
208          Y(I) = 1.0D0
209      ENDDO
210  C
211      CALL FCVREINIT(T, Y, IATOL, RTOL, ATOL, INOPT, IOPT, ROPT, IER)
212      IF (IER .NE. 0) THEN
213          WRITE(6,91) IER
214 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 FCVBBREINIT(NLOCAL, MUDQ, MLDQ, 0.0DO, IER)
222     IF (IER .NE. 0) THEN
223         WRITE(6,92) IER
224 92     FORMAT(///' SUNDIALS_ERROR: FCVBBREINIT returned IER = ', I5)
225         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
226         STOP
227     ENDIF
228 C
229     CALL FCVSPGMRREINIT(IPRE, IGS, 0.0DO, 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

```

```

269     SUBROUTINE FCVGLOCFN(NLOC, T, YLOC, GLOC)
270   C   Routine to define local approximate function g, here the same as f.
271     IMPLICIT NONE
272   C
273     INTEGER*4 NLOC
274     DOUBLE PRECISION YLOC, GLOC, T
275     DIMENSION YLOC(*), GLOC(*)
276   C
277     CALL FCVFUN(T, YLOC, GLOC)
278   C
279     RETURN
280     END
281
282     SUBROUTINE FCVCOMMFN(NLOC, T, YLOC)
283   C   Routine to perform communication required for evaluation of g.
284     RETURN
285     END

```

