

Example Programs for CVODE v2.2.1

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

January 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

| | | |
|----------|---|------------|
| 1 | Introduction | 1 |
| 2 | Serial example problems | 4 |
| 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 |
| 3 | Parallel example problems | 13 |
| 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 |
| 4 | Fortran example problems | 21 |
| 4.1 | A serial example: <code>cvkryf</code> | 21 |
| 4.2 | A parallel example: <code>pvdiagkbf</code> | 23 |
| 5 | Parallel tests | 25 |
| | References | 27 |
| A | Listing of <code>cvdx.c</code> | 28 |
| B | Listing of <code>cvbx.c</code> | 35 |
| C | Listing of <code>cvkx.c</code> | 44 |
| D | Listing of <code>pvnx.c</code> | 57 |
| E | Listing of <code>pvkx.c</code> | 64 |
| F | Listing of <code>pvkxb.c</code> | 83 |
| G | Listing of <code>cvkryf.f</code> | 100 |
| H | Listing of <code>pvdiagkbf.f</code> | 116 |

1 Introduction

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

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

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

- **cvdx** solves a chemical kinetics problem consisting of three rate equations.
This program solves the problem with the BDF method and Newton iteration, with the CVDENSE linear solver and a user-supplied Jacobian routine. It also uses the rootfinding feature of CVODE.
- **cvbx** solves the semi-discrete form of an advection-diffusion equation in 2-D.
This program solves the problem with the BDF method and Newton iteration, with the CVBAND linear solver and a user-supplied Jacobian routine.
- **cvkx** solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D.
The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner. A copy of the block-diagonal part of the Jacobian is saved and conditionally reused within the preconditioner setup routine.
- **cvkxb** solves the same problem as **cvkx**, with the BDF/GMRES method and a banded preconditioner, generated by difference quotients, using the module CVBANDPRE.
The problem is solved twice—with preconditioning on the left, then on the right.
- **cvdemd** is a demonstration program for CVODE with direct linear solvers.
Two separate problems are solved using both the Adams and BDF linear multistep methods in combination with functional and Newton iterations.
The first problem is the Van der Pol oscillator for which the Newton iteration cases use the following types of Jacobian approximations: (1) dense, user-supplied, (2) dense, difference-quotient approximation, (3) diagonal approximation. The second problem is a linear ODE with a banded lower triangular matrix derived from a 2-D advection PDE. In this case, the Newton iteration cases use the following types of Jacobian approximation: (1) banded, user-supplied, (2) banded, difference-quotient approximation, (3) diagonal approximation.
- **cvdemk** is a demonstration program for CVODE with the Krylov linear solver.
This program solves a stiff ODE system that arises from a system of partial differential equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions.
The ODE system is solved using Newton iteration and the CVSPGMR linear solver (scaled preconditioned GMRES).
The preconditioner matrix used is the product of two matrices: (1) a matrix, only

defined implicitly, based on a fixed number of Gauss-Seidel iterations using the diffusion terms only; and (2) a block-diagonal matrix based on the partial derivatives of the interaction terms only, using block-grouping.

Four different runs are made for this problem. The product 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):

- `pvnkx` 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 `sundialtypes.h` file provides the definition of the type `realtype` (see §5.2 for details). For now, it suffices to read `realtype` as `double`. The `cvode.h` file provides prototypes for the CVODE functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `CVode`. The `cvdense.h` file provides the prototype for the `CVDense` function. The `nvector_serial.h` file is the header file for the serial implementation of the `NVECTOR` module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. Finally, the `dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. We have explicitly included `dense.h`, but this is not necessary because it is included by `cvdense.h`.

This program includes two user-defined accessor macros, `Ith` and `IJth` that are useful in writing the problem functions in a form closely matching the mathematical description of the ODE system, i.e. with components numbered from 1 instead of from 0. The `Ith` macro is used to access components of a vector of type `N_Vector` with a serial implementation. It is defined using the `NVECTOR_SERIAL` accessor macro `NV_Ith_S` which numbers components starting with 0. The `IJth` macro is used to access elements of a dense matrix of type `DenseMat`. It is defined using the `DENSE` accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macro `NV_Ith_S` is fully described in §6.1. The macro `DENSE_ELEM` is fully described in §5.6.2.

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

The `main` program begins with some dimensions and type declarations, including use of the type `N_Vector`. The next several lines allocate memory for the `y` and `abstol` vectors using `N_VNew_Serial` with a length argument of `NEQ` ($= 3$). The lines following that load

the initial values of the dependent variable vector into `y` and set the absolute tolerance vector `abstol` using the `Ith` macro.

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

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

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

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

The calls to `CVDense` (see §5.5.2) and `CVDenseSetJacFn` (see §5.5.4) specify the CVDENSE linear solver with an analytic Jacobian supplied by the user-supplied function `Jac`.

The actual solution of the ODE initial value problem is accomplished in the loop over values of the output time `tout`. In each pass of the loop, the program calls `CVode` in the `CV_NORMAL` mode, meaning that the integrator is to take steps until it overshoots `tout` and then interpolate to $t = \text{tout}$, 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 `ydots`. See §5.6.1 for a detailed specification of `f`.

Similarly, the function `g` defines the two functions, g_0 and g_1 , whose roots are to be found. See §5.7.2 for a detailed description of the `g` function.

The function `Jac` sets the nonzero elements of the Jacobian as a dense matrix. (Zero elements need not be set because `J` is preset to zero.) It uses the user-defined macro `IJth` to reference the elements of a dense matrix of type `DenseMat`. Here the problem size is small, so we need not worry about the inefficiency of using `NV_Ith_S` and `DENSE_ELEM` to access `N_Vector` and `DenseMat` elements. Note that in this example, `Jac` only accesses the `y` and `J` arguments. See §5.6.2 for a detailed description of the dense `Jac` function.

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

```

cvdx sample output

3-species kinetics problem

At t = 2.6391e-01      y =  9.899653e-01    3.470564e-05    1.000000e-02
  rootsfound[] =    0    1
At t = 4.0000e-01      y =  9.851641e-01    3.386242e-05    1.480205e-02
At t = 4.0000e+00      y =  9.055097e-01    2.240338e-05    9.446793e-02
At t = 4.0000e+01      y =  7.157952e-01    9.183486e-06    2.841956e-01
At t = 4.0000e+02      y =  4.505420e-01    3.222963e-06    5.494548e-01
At t = 4.0000e+03      y =  1.831878e-01    8.941319e-07    8.168113e-01
At t = 4.0000e+04      y =  3.897868e-02    1.621567e-07    9.610212e-01
At t = 4.0000e+05      y =  4.940023e-03    1.985716e-08    9.950600e-01
At t = 4.0000e+06      y =  5.165107e-04    2.067097e-09    9.994835e-01
At t = 2.0807e+07      y =  1.000000e-04    4.000395e-10    9.999000e-01
  rootsfound[] =    1    0
At t = 4.0000e+07      y =  5.201457e-05    2.080690e-10    9.999480e-01
At t = 4.0000e+08      y =  5.207182e-06    2.082883e-11    9.999948e-01
At t = 4.0000e+09      y =  5.105811e-07    2.042325e-12    9.999995e-01
At t = 4.0000e+10      y =  4.511312e-08    1.804525e-13    1.000000e-00

Final Statistics:
nst = 515    nfe = 754    nsetups = 110    nfeD = 0    njeD = 12
nni = 751    ncnf = 0    netf = 26    nge = 541

```

2.2 A banded example: `cvbx`

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

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

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 $u_{(j-1)+(\text{MY}+1)*i} = v_{ij}$, so that the ODE system is $\dot{u} = f(u)$, then the system Jacobian $J = \partial f / \partial u$ is a band matrix with upper and lower half-bandwidths both equal to MY . In the example, we take $\text{MX} = 10$ 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, MX and MY , the number of equations NEQ , the scalar absolute tolerance ATOL , the initial time T0 , and the initial output time T1 .

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

The type `UserData` is a pointer to a structure containing problem data used in the `f` and `Jac` functions. This structure is allocated and initialized at the beginning of `main`. The pointer to it, called `data`, is passed to both `CVodeSetFData` and `CVBandSetJacData`, and as a result it will be passed back to the `f` and `Jac` functions each time they are called. (If appropriate, two different data structures could be defined and passed to `f` and `Jac`.) The use of the `data` pointer eliminates the need for global program data.

The `main` program is straightforward. The `CVodeCreate` call specifies the `CV_BDF` method with a `CV_NEWTON` iteration. In the `CVodeMalloc` call, the parameter `SS` indicates scalar relative and absolute tolerances, and pointers `&reltol` and `&abstol` to these values are passed. The call to `CVBand` (see §5.5.2) specifies the CVBAND linear solver, and specifies that both half-bandwidths of the Jacobian are equal to MY . The call to `CVBandSetJacFn` (see §5.5.4) specifies that a user-supplied Jacobian function `Jac` is to be used. The actual solution of the problem is performed by the call to `CVode` within the loop over the output times `tout`. The max-norm of the solution vector (from a call to `N_VMaxNorm`) and the cumulative number of time steps (from a call to `CVodeGetNumSteps`) are printed at each output time. Finally, the calls to `PrintFinalStats`, `N_VDestroy`, and `CVodeFree` print statistics and free problem memory.

Following the `main` program in the `cvbx.c` file are definitions of five functions: `f`, `Jac`, `SetIC`, `PrintFinalStats`, and `check_flag`. The last three functions are called only from within the `cvbx.c` file. The `SetIC` function sets the initial dependent variable vector; `PrintFinalStats` gets and prints statistics at the end of the run; and `check_flag` aids in checking return values. The statistics printed include counters such as the total number of steps (`nst`), `f` evaluations (excluding those for 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 `udot`. Note the extra lines setting out-of-bounds values of u to zero.

The `Jac` function is an expression of the derivatives

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

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 to 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 $(i \neq 1)$ etc. in front of the calls to `BAND_COL_ELEM` prevent illegal indexing. See §5.6.3 for a detailed description of the banded `Jac` function.

The output generated by `cvbx` is shown below.

```

_____ cvbx sample output _____

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

At t = 0      max.norm(u) = 8.954716e+01
At t = 0.10   max.norm(u) = 4.132889e+00   nst = 85
At t = 0.20   max.norm(u) = 1.039294e+00   nst = 103
At t = 0.30   max.norm(u) = 2.979829e-01   nst = 113
At t = 0.40   max.norm(u) = 8.765774e-02   nst = 120
At t = 0.50   max.norm(u) = 2.625637e-02   nst = 126
At t = 0.60   max.norm(u) = 7.830425e-03   nst = 130
At t = 0.70   max.norm(u) = 2.329387e-03   nst = 134
At t = 0.80   max.norm(u) = 6.953434e-04   nst = 137
At t = 0.90   max.norm(u) = 2.115983e-04   nst = 140
At t = 1.00   max.norm(u) = 6.556853e-05   nst = 142

Final Statistics:
nst = 142   nfe = 173   nsetups = 23   nfeB = 0   njeB = 3
nni = 170   ncfn = 0   netf = 3

```

2.3 A Krylov example: cvkx

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

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

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

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:

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

where $\omega = \pi/43200$, $a_3 = 22.62$, $a_4 = 7.601$. 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 10×10 mesh, giving an ODE system of size 200.

Among the initial `#include` lines in this case are lines to include `cvspgmr.h` and `sundialsmath.h`. The first contains constants and function prototypes associated with the SPGMR method, including the values of the `pretype` argument to `CVSpgmr`. The inclusion of `sundialsmath.h` is done to access the `SQR` macro for the square of a `realtype` number.

The main program calls `CVodeCreate` specifying the `CV_BDF` method and `CV_NEWTON` iteration, and then calls `CVodeMalloc` with scalar tolerances. It calls `CVSpgmr` (see §5.5.2) to specify the CVSPGMR linear solver with left preconditioning, and the default value (indicated by a zero argument) for `maxl`. The Gram-Schmidt orthogonalization is set to `MODIFIED_GS` through the function `CVSpgmrSetGSType`. Next, user-supplied preconditioner setup and solve functions, `Precond` and `PSolve`, are specified through calls to `CVSpgmrSetPrecSetupFn` and `CVSpgmrSetPrecSolveFn`, respectively. The `data` pointer

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

Then for a sequence of `tout` values, `CVode` is called in the `CV_NORMAL` mode, sampled output is printed, and the return value is tested for error conditions. After that, `PrintFinalStats` is called to get and print final statistics, and memory is freed by calls to `N_VDestroy`, `FreeUserData`, and `CVodeFree`. The printed statistics include various counters, such as the total numbers of steps (`nst`), of `f` evaluations (excluding those for Jv product evaluations) (`nfe`), of `f` evaluations for Jv evaluations (`nfel`), of nonlinear iterations (`nni`), of linear (Krylov) iterations (`nli`), of preconditioner setups (`nsetups`), of preconditioner evaluations (`npe`), and of preconditioner solves (`nps`), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator `CVode`, denoted `lenrw` and `leniw`, and those used by CVSPGMR, denoted `llrw` and `lliw`. All of these optional outputs are described in §5.5.6. The `PrintFinalStats` function is suitable for general use in applications of CVODE to any problem with the SPGMR linear solver.

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

The preconditioner used here is the block-diagonal part of the true Newton matrix. It is generated and factored in the `Precond` routine (see §5.6.6) and backsolved in the `PSolve` routine (see §5.6.5). Its diagonal blocks are 2×2 matrices that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian itself, J_{bd} , is saved in separate storage each time it is generated, on calls to `Precond` with `jok == FALSE`. On calls with `jok == TRUE`, signifying that saved Jacobian data can be reused, the preconditioner $P = I - \gamma J_{bd}$ is formed from the saved matrix J_{bd} and factored. (A call to `Precond` with `jok == TRUE` can only occur after a prior call with `jok == FALSE`.) The `Precond` routine must also set the value of `jcur`, i.e. `*jcurPtr`, to `TRUE` when J_{bd} is re-evaluated, and `FALSE` otherwise, to inform CVSPGMR of the status of Jacobian data.

We need to take a brief detour to explain one last important aspect of the `cvkx.c` program. The generic DENSE solver contains two sets of functions: one for “large” matrices and one for “small” matrices. The large dense functions work with the type `DenseMat`, while the small dense functions work with `realtype **` as the underlying dense matrix types. The CVDENSE linear solver uses the type `DenseMat` for the $N \times N$ dense Jacobian and Newton matrices, and calls the large matrix functions. But to avoid the extra layer of function calls, `cvkx.c` uses the small dense functions for all operations on the 2×2 preconditioner blocks. Thus it includes `smalldense.h`, and calls the small dense matrix functions `denalloc`, `dencopy`, `denscale`, `denaddI`, `denfree`, `denfreepiv`, `gefa`, and `gesl`. The macro `IJth` defined near the top of the file is used to access individual elements in each preconditioner block, numbered from 1. The small dense functions are available for CVODE user programs generally, and are documented in §8.1.

In addition to the functions called by CVODE, `cvkx.c` includes definitions of several private functions. These are: `AllocUserData` to allocate space for J_{bd} , P , and the pivot

arrays; `InitUserData` to load problem constants in the `data` block; `FreeUserData` to free that block; `SetInitialProfiles` to load the initial values in `y`; `PrintOutput` to 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 = 310 | order = 4 | stepsize = 1.80e+02 | |
| c1 (bot.left/middle/top rt.) = | 8.702e+06 | 1.292e+07 | 9.650e+06 | |
| c2 (bot.left/middle/top rt.) = | 3.380e+11 | 5.029e+11 | 3.751e+11 | |
| t = 3.60e+04 | no. steps = 340 | order = 5 | stepsize = 1.15e+02 | |
| c1 (bot.left/middle/top rt.) = | 1.404e+04 | 2.029e+04 | 1.561e+04 | |
| c2 (bot.left/middle/top rt.) = | 3.387e+11 | 4.894e+11 | 3.765e+11 | |
| t = 4.32e+04 | no. steps = 393 | order = 5 | stepsize = 7.13e+02 | |
| c1 (bot.left/middle/top rt.) = | -4.343e-06 | -1.125e-04 | -5.057e-06 | |
| c2 (bot.left/middle/top rt.) = | 3.382e+11 | 1.355e+11 | 3.804e+11 | |
| t = 5.04e+04 | no. steps = 404 | order = 5 | stepsize = 7.37e+02 | |
| c1 (bot.left/middle/top rt.) = | 1.055e-07 | 1.307e-06 | 5.090e-08 | |
| c2 (bot.left/middle/top rt.) = | 3.358e+11 | 4.930e+11 | 3.864e+11 | |
| t = 5.76e+04 | no. steps = 415 | order = 5 | stepsize = 4.24e+02 | |
| c1 (bot.left/middle/top rt.) = | -1.727e-10 | -5.322e-07 | 2.131e-09 | |
| c2 (bot.left/middle/top rt.) = | 3.320e+11 | 9.650e+11 | 3.909e+11 | |
| t = 6.48e+04 | no. steps = 429 | order = 5 | stepsize = 8.83e+02 | |
| c1 (bot.left/middle/top rt.) = | 6.386e-09 | 6.542e-05 | -1.998e-07 | |
| c2 (bot.left/middle/top rt.) = | 3.313e+11 | 8.922e+11 | 3.963e+11 | |
| t = 7.20e+04 | no. steps = 437 | order = 5 | stepsize = 8.83e+02 | |
| c1 (bot.left/middle/top rt.) = | 2.970e-09 | 3.114e-05 | -9.487e-08 | |
| c2 (bot.left/middle/top rt.) = | 3.330e+11 | 6.186e+11 | 4.039e+11 | |
| t = 7.92e+04 | no. steps = 445 | order = 5 | stepsize = 8.83e+02 | |
| c1 (bot.left/middle/top rt.) = | -8.014e-12 | -1.553e-09 | 3.265e-11 | |

```

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

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

```

Final Statistics..

```

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

```


3 Parallel example problems

3.1 A nonstiff example: pvnx

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

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

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 \mathbf{MX} ODEs is obtained by discretizing the x -axis with $\mathbf{MX}+2$ 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 $\Delta x = 2/(\mathbf{MX}+1)$, 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 $i = 1, 2, \dots, \mathbf{MX}$, with the understanding that $u_0 = u_{\mathbf{MX}+1} = 0$.

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 `CNodeGet***` 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 `pvn.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 `pvn` 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.

```

----- pvn sample output -----

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

Number of PEs =    4

At t = 0.00  max.norm(u) =  1.569909e+01  nst =    0
At t = 0.50  max.norm(u) =  3.052881e+00  nst =  113
At t = 1.00  max.norm(u) =  8.753188e-01  nst =  191
At t = 1.50  max.norm(u) =  2.494926e-01  nst =  265
At t = 2.00  max.norm(u) =  7.109707e-02  nst =  339
At t = 2.50  max.norm(u) =  2.026223e-02  nst =  418
At t = 3.00  max.norm(u) =  5.772861e-03  nst =  481
At t = 3.50  max.norm(u) =  1.650209e-03  nst =  551
At t = 4.00  max.norm(u) =  4.718756e-04  nst =  622
At t = 4.50  max.norm(u) =  1.360229e-04  nst =  695
At t = 5.00  max.norm(u) =  4.044654e-05  nst =  761

Final Statistics:

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

```

3.2 A user preconditioner example: pvkx

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

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

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

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

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

```

pvkx sample output

2-species diurnal advection-diffusion problem

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

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

```

```

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

t = 2.88e+04   no. steps = 308   order = 4   stepsize = 2.40e+02
At bottom left: c1, c2 =    8.702e+06    3.380e+11
At top right:   c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 345   order = 4   stepsize = 7.45e+01
At bottom left: c1, c2 =    1.404e+04    3.387e+11
At top right:   c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 402   order = 5   stepsize = 3.33e+02
At bottom left: c1, c2 =    2.995e-06    3.382e+11
At top right:   c1, c2 =    3.316e-06    3.804e+11

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

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

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

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

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

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

```

Final Statistics:

```

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

```

3.3 A CVBBDPRE preconditioner example: pvkxb

In this example, `pvkxb`, we solve the same problem in `pvkx` above, but instead of supplying the preconditioner, we use the CVBBDPRE module, which generates and uses a band-block-diagonal preconditioner. The half-bandwidths of the Jacobian block on each processor are both equal to `2*MXSUB`, and that is the value supplied as `mudq` and `mldq` in the call to `CVBBDPrecAlloc`. But in order to reduce storage and computation costs for preconditioning, we supply the values `mukeep = mlkeep = 2 (= NVARs)` as the half-bandwidths of the retained band matrix blocks. This means that the Jacobian elements are computed with a difference quotient scheme using the true bandwidth of the block, but only a narrow band matrix (bandwidth 5) is kept as the preconditioner. The source is listed in Appendix F.

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

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

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

pvkxb sample output

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

Preconditioner type is:  jpre = PREC_LEFT

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

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

t = 2.16e+04   no. steps = 247   order = 5   stepsize = 3.01e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

t = 2.88e+04   no. steps = 268   order = 5   stepsize = 1.42e+02
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11
```

```

t = 3.60e+04   no. steps = 314   order = 4   stepsize = 8.08e+01
At bottom left: c1, c2 =    1.404e+04    3.387e+11
At top right:   c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 386   order = 4   stepsize = 4.22e+02
At bottom left: c1, c2 =   -2.127e-07    3.382e+11
At top right:   c1, c2 =    3.231e-08    3.804e+11

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

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

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

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

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

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

```

Final Statistics:

```

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

```

```

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

```

```

-----

Preconditioner type is: jpre = PREC_RIGHT

```

```

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

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

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

t = 2.88e+04   no. steps = 322   order = 3   stepsize = 1.09e+02
At bottom left: c1, c2 =    8.702e+06    3.380e+11
At top right:   c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 361   order = 5   stepsize = 7.66e+01
At bottom left: c1, c2 =    1.404e+04    3.387e+11
At top right:   c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 415   order = 5   stepsize = 5.93e+02
At bottom left: c1, c2 =    7.315e-09    3.382e+11
At top right:   c1, c2 =    8.641e-09    3.804e+11

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

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

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

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

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

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

Final Statistics:

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

```

| | | | | | |
|---------|---|-----|------|---|------|
| nfe | = | 652 | nfel | = | 852 |
| nni | = | 649 | nli | = | 852 |
| nsetups | = | 104 | netf | = | 35 |
| npe | = | 9 | nps | = | 1381 |
| ncfn | = | 0 | ncfl | = | 0 |

In CVBBDPRE: real/integer local work space sizes = 600, 50
no. flocl 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 2×2 blocks are LU-factored. In `FCVPSOL`, the solution of a linear system $Px = z$ is solved by doing backsolve operations on the blocks. The remainder of `cvkryf.c` consists of routines from LINPACK and the BLAS needed for matrix and vector operations.

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

```
----- cvkryf sample output -----
Krylov example problem:

Kinetics-transport, NEQ = 200

t = 0.720E+04    no. steps = 219    order = 5    stepsize = 0.158696E+03
c1 (bot.left/middle/top rt.) = 0.104683E+05 0.296373E+05 0.111853E+05
c2 (bot.left/middle/top rt.) = 0.252672E+12 0.715376E+12 0.269977E+12

t = 0.144E+05    no. steps = 251    order = 5    stepsize = 0.377205E+03
c1 (bot.left/middle/top rt.) = 0.665902E+07 0.531602E+07 0.730081E+07
c2 (bot.left/middle/top rt.) = 0.258192E+12 0.205680E+12 0.283286E+12

t = 0.216E+05    no. steps = 277    order = 5    stepsize = 0.274583E+03
```

```

c1 (bot.left/middle/top rt.) = 0.266498E+08 0.103636E+08 0.293077E+08
c2 (bot.left/middle/top rt.) = 0.299279E+12 0.102810E+12 0.331344E+12

t = 0.288E+05 no. steps = 307 order = 4 stepsize = 0.198295E+03
c1 (bot.left/middle/top rt.) = 0.870209E+07 0.129197E+08 0.965002E+07
c2 (bot.left/middle/top rt.) = 0.338035E+12 0.502929E+12 0.375096E+12

t = 0.360E+05 no. steps = 338 order = 5 stepsize = 0.115649E+03
c1 (bot.left/middle/top rt.) = 0.140404E+05 0.202903E+05 0.156090E+05
c2 (bot.left/middle/top rt.) = 0.338677E+12 0.489443E+12 0.376516E+12

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

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

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

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

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

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

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

```

Final statistics:

```

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

```

4.2 A parallel example: pvdiagkbf

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

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

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 `FCVBBDOPT`. To prepare for the second run, the program calls `FCVREINIT`, `FCVBBDREINIT`, and `FCVSPGMRREINIT`, in addition to resetting the initial conditions. Finally, it frees memory and terminates MPI. Notice that in the `FCVFUN` routine, the local processor index `MYPE` and the local vector size `NLOCAL` are used to form the global index values needed to evaluate the right-hand side of Eq. (10).

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

```

pvdiagkbf sample output

Diagonal test problem:

NEQ = 40
parameter alpha = 10.000
ydot_i = -alpha*i * y_i (i = 1,...,NEQ)
RTOL, ATOL = 0.1E-04 0.1E-09
Method is BDF/NEWTON/SPGMR
Preconditioner is band-block-diagonal, using CVBBDPRE
Number of processors = 4

Preconditioning on left

t = 0.10E+00    no. steps = 221    no. f-s = 261
t = 0.20E+00    no. steps = 265    no. f-s = 307
t = 0.30E+00    no. steps = 290    no. f-s = 333
t = 0.40E+00    no. steps = 306    no. f-s = 350
t = 0.50E+00    no. steps = 319    no. f-s = 364
t = 0.60E+00    no. steps = 329    no. f-s = 374
t = 0.70E+00    no. steps = 339    no. f-s = 385
t = 0.80E+00    no. steps = 345    no. f-s = 391
t = 0.90E+00    no. steps = 352    no. f-s = 398
t = 0.10E+01    no. steps = 359    no. f-s = 405

Max. absolute error is 0.28E-08

```

Final statistics:

| | | | | | |
|--|---|-----|------------------------|---|-----|
| number of steps | = | 359 | number of f evals. | = | 405 |
| number of prec. setups | = | 38 | | | |
| number of prec. evals. | = | 7 | number of prec. solves | = | 728 |
| number of nonl. iters. | = | 402 | number of lin. iters. | = | 364 |
| average Krylov subspace dimension (NLI/NNI) = 0.9055 | | | | | |
| number of conv. failures.. nonlinear = 0 linear = 0 | | | | | |
| number of error test failures = 5 | | | | | |

In CVBBDPRE:

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

Preconditioning on right

| | | | | | |
|-----|----------|-------------|-----|-----------|-----|
| t = | 0.10E+00 | no. steps = | 221 | no. f-s = | 261 |
| t = | 0.20E+00 | no. steps = | 265 | no. f-s = | 307 |
| t = | 0.30E+00 | no. steps = | 290 | no. f-s = | 333 |
| t = | 0.40E+00 | no. steps = | 306 | no. f-s = | 350 |
| t = | 0.50E+00 | no. steps = | 319 | no. f-s = | 364 |
| t = | 0.60E+00 | no. steps = | 329 | no. f-s = | 374 |
| t = | 0.70E+00 | no. steps = | 339 | no. f-s = | 385 |
| t = | 0.80E+00 | no. steps = | 345 | no. f-s = | 391 |
| t = | 0.90E+00 | no. steps = | 352 | no. f-s = | 398 |
| t = | 0.10E+01 | no. steps = | 359 | no. f-s = | 405 |

Max. absolute error is 0.28E-08

Final statistics:

| | | | | | |
|--|---|-----|------------------------|---|-----|
| number of steps | = | 359 | number of f evals. | = | 405 |
| number of prec. setups | = | 38 | | | |
| number of prec. evals. | = | 7 | number of prec. solves | = | 728 |
| number of nonl. iters. | = | 402 | number of lin. iters. | = | 364 |
| average Krylov subspace dimension (NLI/NNI) = 0.9055 | | | | | |
| number of conv. failures.. nonlinear = 0 linear = 0 | | | | | |
| number of error test failures = 5 | | | | | |

In CVBBDPRE:

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

5 Parallel tests

The stiff example problem `cvkx` described above, or rather its parallel version `pvkx`, has been modified and expanded to form a test problem for the parallel version of `CVODE`. This work was largely carried out by M. Wittman and reported in [2].

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

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

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 CVOICE 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 CVOICE, with an appropriate choice of MPI implementation, is fully competitive with the Cray-specific version using the SHMEM library. While the overall costs do not prepresent a well-scaled parallel algorithm (because of the preconditioner choice), the cost per function evaluation is quite flat for EPCC and SHMEM, at .033 to .037 (for MPICH it ranges from .044 to .068).

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

References

- [1] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.2.0. Technical Report UCRL-SM-208108, LLNL, 2004.
- [2] M. R. Wittman. Testing of PVODE, a Parallel ODE Solver. Technical Report UCRL-ID-125562, LLNL, August 1996.

A Listing of cvdx.c

```

1  /*
2  * -----
3  * $Revision: 1.19.2.1 $
4  * $Date: 2005/03/17 22:50:36 $
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 "sundialtypes.h" /* definition of type realtype */
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
47 /* User-defined vector and matrix accessor macros: Ith, IJth */
48
49 /* These macros are defined in order to write code which exactly matches
50  the mathematical problem description given above.
51
52  Ith(v,i) references the ith component of the vector v, where i is in

```



```

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

```

```

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

```

```

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

```

```

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

```

```

269     IJth(J,3,2) = RCONST(6.0e7)*y2;
270 }
271
272
273 /*
274  *-----
275  * Private helper functions
276  *-----
277  */
278
279 static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3)
280 {
281     #if defined(SUNDIALS_EXTENDED_PRECISION)
282         printf("At t = %0.4Le      y =%14.6Le  %14.6Le  %14.6Le\n", t, y1, y2, y3);
283     #elif defined(SUNDIALS_DOUBLE_PRECISION)
284         printf("At t = %0.4le      y =%14.6le  %14.6le  %14.6le\n", t, y1, y2, y3);
285     #else
286         printf("At t = %0.4e      y =%14.6e  %14.6e  %14.6e\n", t, y1, y2, y3);
287     #endif
288
289     return;
290 }
291
292 static void PrintRootInfo(int root_f1, int root_f2)
293 {
294     printf("    rootsfound[] = %3d %3d\n", root_f1, root_f2);
295
296     return;
297 }
298
299 /*
300  * Get and print some final statistics
301  */
302
303 static void PrintFinalStats(void *cvode_mem)
304 {
305     long int nst, nfe, nsetups, njeD, nfeD, nni, ncf, netf, nge;
306     int flag;
307
308     flag = CVodeGetNumSteps(cvode_mem, &nst);
309     check_flag(&flag, "CVodeGetNumSteps", 1);
310     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
311     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
312     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
313     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
314     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
315     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
316     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
317     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
318     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncf);
319     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
320
321     flag = CVDenseGetNumJacEvals(cvode_mem, &njeD);
322     check_flag(&flag, "CVDenseGetNumJacEvals", 1);

```

```

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

```

B Listing of cvbx.c

```

1  /*
2  * -----
3  * $Revision: 1.17.2.1 $
4  * $Date: 2005/03/17 22:50:36 $
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         T0         is the initial time
160         u          is the initial dependent variable vector

```

```

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

```

```

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

```

```

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

```

```

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

```

```

377     check_flag(&flag, "CVodeGetNumSteps", 1);
378     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
379     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
380     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
381     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
382     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
383     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
384     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
385     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
386     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
387     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
388
389     flag = CVBandGetNumJacEvals(cvode_mem, &njeB);
390     check_flag(&flag, "CVBandGetNumJacEvals", 1);
391     flag = CVBandGetNumRhsEvals(cvode_mem, &nfeB);
392     check_flag(&flag, "CVBandGetNumRhsEvals", 1);
393
394     printf("\nFinal Statistics:\n");
395     printf("nst = %-6ld nfe = %-6ld nsetups = %-6ld nfeB = %-6ld njeB = %ld\n",
396           nst, nfe, nsetups, nfeB, njeB);
397     printf("nni = %-6ld ncnf = %-6ld netf = %ld\n \n",
398           nni, ncnf, netf);
399
400     return;
401 }
402
403 /* Check function return value...
404     opt == 0 means SUNDIALS function allocates memory so check if
405         returned NULL pointer
406     opt == 1 means SUNDIALS function returns a flag so check if
407         flag >= 0
408     opt == 2 means function allocates memory so check if returned
409         NULL pointer */
410
411 static int check_flag(void *flagvalue, char *funcname, int opt)
412 {
413     int *errflag;
414
415     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
416
417     if (opt == 0 && flagvalue == NULL) {
418         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
419             funcname);
420         return(1); }
421
422     /* Check if flag < 0 */
423
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  /* Check if function returned NULL pointer - no memory allocated */
432
433  else if (opt == 2 && flagvalue == NULL) {
434      fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
435              funcname);
436      return(1); }
437
438  return(0);
439  }

```

C Listing of cvkx.c

```

1  /*
2  * -----
3  * $Revision: 1.17.2.1 $
4  * $Date: 2005/03/17 22:50:36 $
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 realtype, 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     ccode_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 CcodeCreate 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 ccode_mem. */
181     ccode_mem = CcodeCreate(CV_BDF, CV_NEWTON);
182     if(check_flag((void *)ccode_mem, "CcodeCreate", 0)) return(1);
183
184     /* Set the pointer to user-defined data */
185     flag = CcodeSetFdata(ccode_mem, data);
186     if(check_flag(&flag, "CcodeSetFdata", 1)) return(1);
187
188     /* Call CcodeMalloc to initialize the integrator memory:
189
190         f          is the user's right hand side function in u'=f(t,u)
191         T0         is the initial time
192         u          is the initial dependent variable vector
193         CV_SS      specifies scalar relative and absolute tolerances
194         &reltol and &abstol are pointers to the scalar tolerances      */
195     flag = CcodeMalloc(ccode_mem, f, T0, u, CV_SS, &reltol, &abstol);
196     if(check_flag(&flag, "CcodeMalloc", 1)) return(1);
197
198     /* Call CVSpgrmr to specify the linear solver CVSPGMR
199         with left preconditioning and the maximum Krylov dimension maxl */
200     flag = CVSpgrmr(ccode_mem, PREC_LEFT, 0);
201     if(check_flag(&flag, "CVSpgrmr", 1)) return(1);
202
203     /* Set modified Gram-Schmidt orthogonalization, preconditioner
204         setup and solve routines Precond and PSolve, and the pointer
205         to the user-defined block data */
206     flag = CVSpgrmrSetGSType(ccode_mem, MODIFIED_GS);
207     if(check_flag(&flag, "CVSpgrmrSetGSType", 1)) return(1);
208
209     flag = CVSpgrmrSetPrecSetupFn(ccode_mem, Precond);
210     if(check_flag(&flag, "CVSpgrmrSetPrecSetupFn", 1)) return(1);
211
212     flag = CVSpgrmrSetPrecSolveFn(ccode_mem, PSolve);
213     if(check_flag(&flag, "CVSpgrmrSetPrecSolveFn", 1)) return(1);
214

```

```

215     flag = CVSpgmrSetPrecData(cvode_mem, data);
216     if(check_flag(&flag, "CVSpgmrSetPrecData", 1)) return(1);
217
218     /* In loop over output points, call CVode, print results, test for error */
219     printf(" \n2-species diurnal advection-diffusion problem\n\n");
220     for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
221         flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
222         PrintOutput(cvode_mem, u, t);
223         if(check_flag(&flag, "CVode", 1)) break;
224     }
225
226     PrintFinalStats(cvode_mem);
227
228     /* Free memory */
229     N_VDestroy_Serial(u);
230     FreeUserData(data);
231     CVodeFree(cvode_mem);
232
233     return(0);
234 }
235
236 /*
237 *-----
238 * Private helper functions
239 *-----
240 */
241
242 /* Allocate memory for data structure of type UserData */
243
244 static UserData AllocUserData(void)
245 {
246     int jx, jy;
247     UserData data;
248
249     data = (UserData) malloc(sizeof *data);
250
251     for (jx=0; jx < MX; jx++) {
252         for (jy=0; jy < MY; jy++) {
253             (data->P)[jx][jy] = denalloc(NUM_SPECIES);
254             (data->Jbd)[jx][jy] = denalloc(NUM_SPECIES);
255             (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
256         }
257     }
258
259     return(data);
260 }
261
262 /* Load problem constants in data */
263
264 static void InitUserData(UserData data)
265 {
266     data->om = PI/HALFDAY;
267     data->dx = (XMAX-XMIN)/(MX-1);
268     data->dy = (YMAX-YMIN)/(MY-1);

```

```

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

```

```

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

```

```

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

```

```

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

```



```

485
486 /* Loop over all grid points. */
487
488 for (jy=0; jy < MY; jy++) {
489
490     /* Set vertical diffusion coefficients at jy +- 1/2 */
491
492     ydn = YMIN + (jy - RCONST(0.5))*dely;
493     yup = ydn + dely;
494     cydn = verdco*exp(RCONST(0.2)*ydn);
495     cyup = verdco*exp(RCONST(0.2)*yup);
496     idn = (jy == 0) ? 1 : -1;
497     iup = (jy == MY-1) ? -1 : 1;
498     for (jx=0; jx < MX; jx++) {
499
500         /* Extract c1 and c2, and set kinetic rate terms. */
501
502         c1 = IJKth(udata,1,jx,jy);
503         c2 = IJKth(udata,2,jx,jy);
504         qq1 = Q1*c1*C3;
505         qq2 = Q2*c1*c2;
506         qq3 = q3*C3;
507         qq4 = q4coef*c2;
508         rkin1 = -qq1 - qq2 + TW0*qq3 + qq4;
509         rkin2 = qq1 - qq2 - qq4;
510
511         /* Set vertical diffusion terms. */
512
513         c1dn = IJKth(udata,1,jx,jy+idn);
514         c2dn = IJKth(udata,2,jx,jy+idn);
515         c1up = IJKth(udata,1,jx,jy+iup);
516         c2up = IJKth(udata,2,jx,jy+iup);
517         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
518         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
519
520         /* Set horizontal diffusion and advection terms. */
521
522         ileft = (jx == 0) ? 1 : -1;
523         irtight = (jx == MX-1) ? -1 : 1;
524         c1lt = IJKth(udata,1,jx+ileft,jy);
525         c2lt = IJKth(udata,2,jx+ileft,jy);
526         c1rt = IJKth(udata,1,jx+irtight,jy);
527         c2rt = IJKth(udata,2,jx+irtight,jy);
528         hord1 = hordco*(c1rt - TW0*c1 + c1lt);
529         hord2 = hordco*(c2rt - TW0*c2 + c2lt);
530         horad1 = horaco*(c1rt - c1lt);
531         horad2 = horaco*(c2rt - c2lt);
532
533         /* Load all terms into udot. */
534
535         IJKth(dudata, 1, jx, jy) = vertd1 + hord1 + horad1 + rkin1;
536         IJKth(dudata, 2, jx, jy) = vertd2 + hord2 + horad2 + rkin2;
537     }
538 }

```

```

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

```

```

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

```

```

647 P = data->P;
648 pivot = data->pivot;
649 zdata = NV_DATA_S(z);
650
651 N_VScale(ONE, r, z);
652
653 /* Solve the block-diagonal system Px = r using LU factors stored
654    in P and pivot data in pivot, and return the solution in z. */
655
656 for (jx=0; jx < MX; jx++) {
657     for (jy=0; jy < MY; jy++) {
658         v = &(IJKth(zdata, 1, jx, jy));
659         gesl(P[jx][jy], NUM_SPECIES, pivot[jx][jy], v);
660     }
661 }
662
663 return(0);
664 }

```

D Listing of pvn.c

```
1  /*
2  * -----
3  * $Revision: 1.12.2.1 $
4  * $Date: 2005/03/17 22:50:31 $
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 realtype */
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 *cnode_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 *cnode_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    cnode_mem = NULL;
101
102    /* Get processor number, total number of pe's, and my_pe. */
103    MPI_Init(&argc, &argv);
104    comm = MPI_COMM_WORLD;
105    MPI_Comm_size(comm, &npes);
106    MPI_Comm_rank(comm, &my_pe);

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

E Listing of pvkx.c

```

1  /*
2  * -----
3  * $Revision: 1.14.2.1 $
4  * $Date: 2005/03/17 22:50:31 $
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 realtype, booleantype, TRUE, */
55                             /* and FALSE */
56 #include "sundialsmath.h" /* definition of macro SQR */
57 #include "cvmcode.h" /* prototypes for Cvmode* 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 NVAR 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 /* CvmodeMalloc 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 dsizex, long int dsizey,
149     realtype udata[]);
150 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
151     int my_pe, int isubx, int isuby,
152     long int dsizex, long int dsizey,
153     realtype uext[], realtype buffer[]);
154 static void BRecvWait(MPI_Request request[],
155     int isubx, int isuby,
156     long int dsizex, realtype uext[],
157     realtype buffer[]);
158 static void ucomm(realtype t, N_Vector u, UserData data);
159 static void fcalc(realtype t, realtype udata[], realtype dudata[],
160     UserData data);

```

```

161
162
163 /* Functions Called by the Solver */
164
165 static void f(realtype t, N_Vector u, N_Vector udot, void *f_data);
166
167 static int Precond(realtype tn, N_Vector u, N_Vector fu,
168                   booleantype jok, booleantype *jcurPtr,
169                   realtype gamma, void *P_data,
170                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);
171
172 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
173                  N_Vector r, N_Vector z,
174                  realtype gamma, realtype delta,
175                  int lr, void *P_data, N_Vector vtemp);
176
177
178 /* Private function to check function return values */
179
180 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
181
182
183 /***** Main Program *****/
184
185 int main(int argc, char *argv[])
186 {
187     realtype abstol, reltol, t, tout;
188     N_Vector u;
189     UserData data;
190     PreconData predata;
191     void *cnode_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     cnode_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\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 CNodeCreate 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 cnode_mem.
240 */
241 cnode_mem = CNodeCreate(CV_BDF, CV_NEWTON);
242 if (check_flag((void *)cnode_mem, "CNodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
243
244 /* Set the pointer to user-defined data */
245 flag = CNodeSetFdata(cnode_mem, data);
246 if (check_flag(&flag, "CNodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
247
248 /*
249     Call CNodeMalloc to initialize the integrator memory:
250
251     cnode_mem is the pointer to the integrator memory returned by CNodeCreate
252     f          is the user's right hand side function in  $y'=f(t,y)$ 
253     T0         is the initial time
254     u          is the initial dependent variable vector
255     CV_SS      specifies scalar relative and absolute tolerances
256     &reltol and &abstol are pointers to the scalar tolerances
257 */
258 flag = CNodeMalloc(cnode_mem, f, T0, u, CV_SS, &reltol, &abstol);
259 if (check_flag(&flag, "CNodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
260
261 /* Call CVSpgrmr to specify the linear solver CVSPGMR
262     with left preconditioning and the maximum Krylov dimension maxl */
263 flag = CVSpgrmr(cnode_mem, PREC_LEFT, 0);
264 if (check_flag(&flag, "CVSpgrmr", 1, my_pe)) MPI_Abort(comm, 1);
265
266 /* Set preconditioner setup and solve routines Precond and PSolve,
267     and the pointer to the user-defined block data */
268 flag = CVSpgrmrSetPrecSetupFn(cnode_mem, Precond);

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

593     }
594
595     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
596     if (isubx != NPEX-1) {
597         MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
598                 my_pe+1, 0, comm, &request[3]);
599     }
600 }
601
602 /* Routine to finish receiving boundary data from neighboring PEs.
603    Notes:
604    1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
605    passed to both the BRecvPost and BRecvWait functions, and should not
606    be manipulated between the two calls.
607    2) request should have 4 entries, and should be passed in both calls also. */
608
609 static void BRecvWait(MPI_Request request[],
610                      int isubx, int isuby,
611                      long int dsizey, realtype uext[],
612                      realtype buffer[])
613 {
614     int i, ly;
615     long int dsizey2, offsetue, offsetbuf;
616     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
617     MPI_Status status;
618
619     dsizey2 = dsizey + 2*NVARs;
620
621     /* If isuby > 0, receive data for bottom x-line of uext */
622     if (isuby != 0)
623         MPI_Wait(&request[0], &status);
624
625     /* If isuby < NPEY-1, receive data for top x-line of uext */
626     if (isuby != NPEY-1)
627         MPI_Wait(&request[1], &status);
628
629     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
630     if (isubx != 0) {
631         MPI_Wait(&request[2], &status);
632
633         /* Copy the buffer to uext */
634         for (ly = 0; ly < MYSUB; ly++) {
635             offsetbuf = ly*NVARs;
636             offsetue = (ly+1)*dsizey2;
637             for (i = 0; i < NVARs; i++)
638                 uext[offsetue+i] = bufleft[offsetbuf+i];
639         }
640     }
641
642     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
643     if (isubx != NPEX-1) {
644         MPI_Wait(&request[3], &status);
645
646         /* Copy the buffer to uext */

```

```

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

```



```

701 long int nvmsub, nvmsub2, offsetu, offsetue;
702
703 /* Get subgrid indices, data sizes, extended work array uext */
704 isubx = data->isubx; isuby = data->isuby;
705 nvmsub = data->nvmsub; nvmsub2 = data->nvmsub2;
706 uext = data->uext;
707
708 /* Copy local segment of u vector into the working extended array uext */
709 offsetu = 0;
710 offsetue = nvmsub2 + NVAR;
711 for (ly = 0; ly < MYSUB; ly++) {
712     for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
713     offsetu = offsetu + nvmsub;
714     offsetue = offsetue + nvmsub2;
715 }
716
717 /* To facilitate homogeneous Neumann boundary conditions, when this is
718 a boundary PE, copy data from the first interior mesh line of u to uext */
719
720 /* If isuby = 0, copy x-line 2 of u to uext */
721 if (isuby == 0) {
722     for (i = 0; i < nvmsub; i++) uext[NVAR+i] = udata[nvmsub+i];
723 }
724
725 /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
726 if (isuby == NPEY-1) {
727     offsetu = (MYSUB-2)*nvmsub;
728     offsetue = (MYSUB+1)*nvmsub2 + NVAR;
729     for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
730 }
731
732 /* If isubx = 0, copy y-line 2 of u to uext */
733 if (isubx == 0) {
734     for (ly = 0; ly < MYSUB; ly++) {
735         offsetu = ly*nvmsub + NVAR;
736         offsetue = (ly+1)*nvmsub2;
737         for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
738     }
739 }
740
741 /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
742 if (isubx == NPEX-1) {
743     for (ly = 0; ly < MYSUB; ly++) {
744         offsetu = (ly+1)*nvmsub - 2*NVAR;
745         offsetue = (ly+2)*nvmsub2 - NVAR;
746         for (i = 0; i < NVAR; i++) uext[offsetue+i] = udata[offsetu+i];
747     }
748 }
749
750 /* Make local copies of problem variables, for efficiency */
751 dely = data->dy;
752 verdco = data->vdco;
753 hordco = data->hdco;
754 horaco = data->haco;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

F Listing of pvkxb.c

```

1  /*
2  * -----
3  * $Revision: 1.19.2.1 $
4  * $Date: 2005/03/17 22:50:31 $
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 "cvscode.h"           /* prototypes for CNode* 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 /* CNodeMalloc 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 nvmsub, nvmsub2, 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 dsizex, 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 dsizex, long int dsizey,
139                      realtype uext[], realtype buffer[]);
140 static void BRecvWait(MPI_Request request[],
141                      int isubx, int isuby,
142                      long int dsizex, 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 ccode_mem.
216 */
217
218 ccode_mem = CCodeCreate(CV_BDF, CV_NEWTON);
219 if(check_flag((void *)ccode_mem, "CCodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
220
221 /* Set the pointer to user-defined data */
222 flag = CCodeSetFdata(ccode_mem, data);
223 if(check_flag(&flag, "CCodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
224
225 /*
226     Call CCodeMalloc to initialize the integrator memory:
227
228     ccode_mem is the pointer to the integrator memory returned by CCodeCreate
229     f         is the user's right hand side function in  $y'=f(t,y)$ 
230     T0        is the initial time
231     u         is the initial dependent variable vector
232     CV_SS     specifies scalar relative and absolute tolerances
233     &reltol and &abstol are pointers to the scalar tolerances
234 */
235
236 flag = CCodeMalloc(ccode_mem, f, T0, u, CV_SS, &reltol, &abstol);
237 if(check_flag(&flag, "CCodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
238
239 /* Allocate preconditioner block */
240 mudq = mldq = NCVARS*MXSUB;
241 mkeep = mlkeep = NCVARS;
242 pdata = CVBBDPrecAlloc(ccode_mem, local_N, mudq, mldq,
243                        mkeep, mlkeep, ZERO, flocal, NULL);
244 if(check_flag((void *)pdata, "CVBBDPrecAlloc", 0, my_pe)) MPI_Abort(comm, 1);
245
246 /* Call CVBBDSPgmr to specify the linear solver CVSPGMR using the
247     CVBBDPRE preconditioner, with left preconditioning and the
248     default maximum Krylov dimension maxl */
249 flag = CVBBDSPgmr(ccode_mem, PREC_LEFT, 0, pdata);
250 if(check_flag(&flag, "CVBBDSPgmr", 1, my_pe)) MPI_Abort(comm, 1);
251
252 /* Print heading */
253 if (my_pe == 0) PrintIntro(npes, mudq, mldq, mkeep, mlkeep);
254
255 /* Loop over jpre (= PREC_LEFT, PREC_RIGHT), and solve the problem */
256 for (jpre = PREC_LEFT; jpre <= PREC_RIGHT; jpre++) {
257
258     /* On second run, re-initialize u, the integrator, CVBBDPRE, and CVSPGMR */
259
260     if (jpre == PREC_RIGHT) {
261
262         SetInitialProfiles(u, data);
263
264         flag = CCodeReInit(ccode_mem, f, T0, u, CV_SS, &reltol, &abstol);
265         if(check_flag(&flag, "CCodeReInit", 1, my_pe)) MPI_Abort(comm, 1);
266
267         flag = CVBBDPrecReInit(pdata, mudq, mldq, ZERO, flocal, NULL);
268         if(check_flag(&flag, "CVBBDPrecReInit", 1, my_pe)) MPI_Abort(comm, 1);

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

593     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
594               my_pe-1, 0, comm, &request[2]);
595 }
596
597 /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
598 if (isubx != NPEX-1) {
599     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
600               my_pe+1, 0, comm, &request[3]);
601 }
602
603 }
604
605 /* Routine to finish receiving boundary data from neighboring PEs.
606    Notes:
607    1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
608    passed to both the BRecvPost and BRecvWait functions, and should not
609    be manipulated between the two calls.
610    2) request should have 4 entries, and should be passed in both calls also. */
611
612 static void BRecvWait(MPI_Request request[],
613                       int isubx, int isuby,
614                       long int dsizey, realtype uext[],
615                       realtype buffer[])
616 {
617     int i, ly;
618     long int dsizey2, offsetue, offsetbuf;
619     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
620     MPI_Status status;
621
622     dsizey2 = dsizey + 2*NVARs;
623
624     /* If isuby > 0, receive data for bottom x-line of uext */
625     if (isuby != 0)
626         MPI_Wait(&request[0], &status);
627
628     /* If isuby < NPEY-1, receive data for top x-line of uext */
629     if (isuby != NPEY-1)
630         MPI_Wait(&request[1], &status);
631
632     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
633     if (isubx != 0) {
634         MPI_Wait(&request[2], &status);
635
636         /* Copy the buffer to uext */
637         for (ly = 0; ly < MYSUB; ly++) {
638             offsetbuf = ly*NVARs;
639             offsetue = (ly+1)*dsizey2;
640             for (i = 0; i < NVARs; i++)
641                 uext[offsetue+i] = bufleft[offsetbuf+i];
642         }
643     }
644
645     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
646     if (isubx != NPEX-1) {

```

```

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

```

```

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

```

```

755  /* To facilitate homogeneous Neumann boundary conditions, when this is
756  a boundary PE, copy data from the first interior mesh line of u to uext */
757
758  /* If isuby = 0, copy x-line 2 of u to uext */
759  if (isuby == 0) {
760      for (i = 0; i < nvmsub; i++) uext[NVARS+i] = uarray[nvmsub+i];
761  }
762
763  /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
764  if (isuby == NPEY-1) {
765      offsetu = (MYSUB-2)*nvmsub;
766      offsetue = (MYSUB+1)*nvmsub2 + NVARS;
767      for (i = 0; i < nvmsub; i++) uext[offsetue+i] = uarray[offsetu+i];
768  }
769
770  /* If isubx = 0, copy y-line 2 of u to uext */
771  if (isubx == 0) {
772      for (ly = 0; ly < MYSUB; ly++) {
773          offsetu = ly*nvmsub + NVARS;
774          offsetue = (ly+1)*nvmsub2;
775          for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
776      }
777  }
778
779  /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
780  if (isubx == NPEX-1) {
781      for (ly = 0; ly < MYSUB; ly++) {
782          offsetu = (ly+1)*nvmsub - 2*NVARS;
783          offsetue = (ly+2)*nvmsub2 - NVARS;
784          for (i = 0; i < NVARS; i++) uext[offsetue+i] = uarray[offsetu+i];
785      }
786  }
787
788  /* Make local copies of problem variables, for efficiency */
789
790  dely = data->dy;
791  verdco = data->vdco;
792  hordco = data->hdco;
793  horaco = data->haco;
794
795  /* Set diurnal rate coefficients as functions of t, and save q4 in
796  data block for use by preconditioner evaluation routine */
797
798  s = sin((data->om)*t);
799  if (s > ZERO) {
800      q3 = exp(-A3/s);
801      q4coef = exp(-A4/s);
802  } else {
803      q3 = ZERO;
804      q4coef = ZERO;
805  }
806  data->q4 = q4coef;
807
808

```

```

809  /* Loop over all grid points in local subgrid */
810
811  for (ly = 0; ly < MYSUB; ly++) {
812
813      jy = ly + isuby*MYSUB;
814
815      /* Set vertical diffusion coefficients at jy +- 1/2 */
816
817      ydn = YMIN + (jy - RCONST(0.5))*dely;
818      yup = ydn + dely;
819      cydn = verdco*exp(RCONST(0.2)*ydn);
820      cyup = verdco*exp(RCONST(0.2)*yup);
821      for (lx = 0; lx < MXSUB; lx++) {
822
823          jx = lx + isubx*MXSUB;
824
825          /* Extract c1 and c2, and set kinetic rate terms */
826
827          offsetue = (lx+1)*NVARs + (ly+1)*nvmsub2;
828          c1 = uext[offsetue];
829          c2 = uext[offsetue+1];
830          qq1 = Q1*c1*C3;
831          qq2 = Q2*c1*c2;
832          qq3 = q3*C3;
833          qq4 = q4coef*c2;
834          rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
835          rkin2 = qq1 - qq2 - qq4;
836
837          /* Set vertical diffusion terms */
838
839          c1dn = uext[offsetue-nvmsub2];
840          c2dn = uext[offsetue-nvmsub2+1];
841          c1up = uext[offsetue+nvmsub2];
842          c2up = uext[offsetue+nvmsub2+1];
843          vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
844          vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
845
846          /* Set horizontal diffusion and advection terms */
847
848          c1lt = uext[offsetue-2];
849          c2lt = uext[offsetue-1];
850          c1rt = uext[offsetue+2];
851          c2rt = uext[offsetue+3];
852          hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
853          hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
854          horad1 = horaco*(c1rt - c1lt);
855          horad2 = horaco*(c2rt - c2lt);
856
857          /* Load all terms into duarray */
858
859          offsetu = lx*NVARs + ly*nvmsub;
860          duarray[offsetu] = vertd1 + hord1 + horad1 + rkin1;
861          duarray[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
862      }

```

```

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

```

G Listing of cvkryf.f

```

1 C -----
2 C $Revision: 1.20 $
3 C $Date: 2004/11/22 23:23:27 $
4 C -----
5 C FCVODE Example Problem: 2D kinetics-transport, 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 DCOPY and DSCAL.
31 C
32 C The second and third dimensions of U here must match the values
33 C of MESHX and MESHY, for consistency with the output statements
34 C below.
35 C -----
36 C
37 C IMPLICIT NONE
38 C
39 C INTEGER METH, ITMETH, IATOL, INOPT, ITASK, IER, LNCFL, LNPS
40 C INTEGER LNST, LNFE, LNSETUP, LNNI, 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/, LNNI/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 FCVSPGMRSETPSET(1, IER)
99  C
100     CALL FCVSPGMRSETPSOL(1, IER)
101  C
102  C Loop over output points, call FCVODE, print sample solution values.
103      TOUT = TWOHR
104      DO 70 IOUT = 1, 12
105  C
106          CALL FCVODE(TOUT, T, U, ITASK, IER)

```

```

107 C
108     WRITE(6,50) T, IOPT(LNST), IOPT(LQ), ROPT(LH)
109 50     FORMAT(/' t = ', E11.3, 5X, 'no. steps = ', I5,
110 1       '    order = ', I3, '    stepsize = ', E14.6)
111     WRITE(6,55) U(1,1,1), U(1,5,5), U(1,10,10),
112 1       U(2,1,1), U(2,5,5), U(2,10,10)
113 55     FORMAT('  c1 (bot.left/middle/top rt.) = ', 3E14.6/
114 1       '  c2 (bot.left/middle/top rt.) = ', 3E14.6)
115 C
116     IF (IER .NE. 0) THEN
117         WRITE(6,60) IER, IOPT(26)
118 60     FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
119 1         '                               Linear Solver returned IER = ', I5)
120         CALL FNVFREES
121         CALL FCVFREE
122         STOP
123     ENDIF
124 C
125     TOUT = TOUT + TWOHR
126 70     CONTINUE
127
128 C Print final statistics.
129     NST = IOPT(LNST)
130     NFE = IOPT(LNFE)
131     NPSET = IOPT(LNSETUP)
132     NPE = IOPT(LNPE)
133     NPS = IOPT(LNPS)
134     NNI = IOPT(LNNI)
135     NLI = IOPT(LNLI)
136     AVDIM = DBLE(NLI) / DBLE(NNI)
137     NCFN = IOPT(LNCF)
138     NCFL = IOPT(LNCFL)
139     WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
140 1     NCFL
141 80     FORMAT(///'Final statistics: '//
142 1     ' number of steps          = ', I5, 5X,
143 2     'number of f evals.       = ', I5/,
144 3     ' number of prec. setups = ', I5/,
145 4     ' number of prec. evals. = ', I5, 5X,
146 5     'number of prec. solves = ', I5/,
147 6     ' number of nonl. iters. = ', I5, 5X,
148 7     'number of lin. iters.  = ', I5/,
149 8     ' average Krylov subspace dimension (NLI/NNI) = ', E14.6/
150 9     ' number of conv. failures.. nonlinear = ', I3,' linear = ', I3)
151 C
152     CALL FCVFREE
153     CALL FNVFREES
154 C
155     STOP
156     END
157
158     SUBROUTINE INITKX(MESHX, MESHY, U0)
159 C Routine to set problem constants and initial values
160 C

```

```

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

```

```

215     INTEGER ILEFT, IRIGHT
216     INTEGER*4 JX, JY, MX, MY, MM, IBLOK0, IBLOK, IDN, IUP
217     DOUBLE PRECISION T, U(2,*), UDOT(2,*)
218     DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
219     DOUBLE PRECISION VDCO, HACO
220     DOUBLE PRECISION C1, C2, C1DN, C2DN, C1UP, C2UP, C1LT, C2LT
221     DOUBLE PRECISION C1RT, C2RT, CYDN, CYUP, HORD1, HORD2, HORAD1
222     DOUBLE PRECISION HORAD2, QQ1, QQ2, QQ3, QQ4, RKIN1, RKIN2, S
223     DOUBLE PRECISION VERTD1, VERTD2, YDN, YUP
224 C
225     COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
226     COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
227 C
228 C Set diurnal rate coefficients.
229     S = SIN(OM * T)
230     IF (S .GT. 0.0D0) THEN
231         Q3 = EXP(-A3 / S)
232         Q4 = EXP(-A4 / S)
233     ELSE
234         Q3 = 0.0D0
235         Q4 = 0.0D0
236     ENDIF
237 C
238 C Loop over all grid points.
239     DO 20 JY = 1, MY
240         YDN = 30.0D0 + (JY - 1.5D0) * DY
241         YUP = YDN + DY
242         CYDN = VDCO * EXP(0.2D0 * YDN)
243         CYUP = VDCO * EXP(0.2D0 * YUP)
244         IBLOK0 = (JY - 1) * MX
245         IDN = -MX
246         IF (JY .EQ. 1) IDN = MX
247         IUP = MX
248         IF (JY .EQ. MY) IUP = -MX
249         DO 10 JX = 1, MX
250             IBLOK = IBLOK0 + JX
251             C1 = U(1,IBLOK)
252             C2 = U(2,IBLOK)
253 C Set kinetic rate terms.
254             QQ1 = Q1 * C1 * C3
255             QQ2 = Q2 * C1 * C2
256             QQ3 = Q3 * C3
257             QQ4 = Q4 * C2
258             RKIN1 = -QQ1 - QQ2 + 2.0D0 * QQ3 + QQ4
259             RKIN2 = QQ1 - QQ2 - QQ4
260 C Set vertical diffusion terms.
261             C1DN = U(1,IBLOK + IDN)
262             C2DN = U(2,IBLOK + IDN)
263             C1UP = U(1,IBLOK + IUP)
264             C2UP = U(2,IBLOK + IUP)
265             VERTD1 = CYUP * (C1UP - C1) - CYDN * (C1 - C1DN)
266             VERTD2 = CYUP * (C2UP - C2) - CYDN * (C2 - C2DN)
267 C Set horizontal diffusion and advection terms.
268             ILEFT = -1

```

```

269         IF (JX .EQ. 1) ILEFT = 1
270         IRIGHT = 1
271         IF (JX .EQ. MX) IRIGHT = -1
272         C1LT = U(1,IBLOK + ILEFT)
273         C2LT = U(2,IBLOK + ILEFT)
274         C1RT = U(1,IBLOK + IRIGHT)
275         C2RT = U(2,IBLOK + IRIGHT)
276         HORD1 = HDCO * (C1RT - 2.0D0 * C1 + C1LT)
277         HORD2 = HDCO * (C2RT - 2.0D0 * C2 + C2LT)
278         HORAD1 = HACO * (C1RT - C1LT)
279         HORAD2 = HACO * (C2RT - C2LT)
280 C Load all terms into UDOT.
281         UDOT(1,IBLOK) = VERTD1 + HORD1 + HORAD1 + RKIN1
282         UDOT(2,IBLOK) = VERTD2 + HORD2 + HORAD2 + RKIN2
283     10     CONTINUE
284     20     CONTINUE
285     RETURN
286     END
287
288     SUBROUTINE FCVPSET(T, U, FU, JOK, JCUR, GAMMA, EWT, H,
289     1         V1, V2, V3, IER)
290 C Routine to set and preprocess block-diagonal preconditioner.
291 C Note: The dimensions in /BDJ/ below assume at most 100 mesh points.
292 C
293     IMPLICIT NONE
294 C
295     INTEGER IER, JOK, JCUR, H
296     INTEGER*4 LENBD, JY, JX, IBLOK, MX, MY, MM
297     INTEGER*4 IBLOK0, IPP
298     DOUBLE PRECISION T, U(2,*), GAMMA
299     DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
300     DOUBLE PRECISION VDCO, HACO
301     DOUBLE PRECISION BD, P, FU, EWT, V1, V2, V3
302     DOUBLE PRECISION C1, C2, CYDN, CYUP, DIAG, TEMP, YDN, YUP
303 C
304     COMMON /PCOM/ Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY
305     COMMON /PCOM/ HDCO, VDCO, HACO, MX, MY, MM
306     COMMON /BDJ/ BD(2,2,100), P(2,2,100), IPP(2,100)
307 C
308     IER = 0
309     LENBD = 4 * MM
310 C
311 C If JOK = 1, copy BD to P.
312     IF (JOK .EQ. 1) THEN
313         CALL DCOPY(LENBD, BD(1,1,1), 1, P(1,1,1), 1)
314         JCUR = 0
315     ELSE
316 C
317 C JOK = 0. Compute diagonal Jacobian blocks and copy to P.
318 C (using q4 value computed on last FCVFUN call).
319         DO 20 JY = 1, MY
320             YDN = 30.0D0 + (JY - 1.5D0) * DY
321             YUP = YDN + DY
322             CYDN = VDCO * EXP(0.2D0 * YDN)

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

809      if (n .le. 0) return
810      if (incx .eq. 1 .and. incy .eq. 1) go to 20
811  c
812  c          code for unequal increments or equal increments
813  c          not equal to 1
814  c
815      ix = 1
816      iy = 1
817      if (incx .lt. 0) ix = (-n + 1) * incx + 1
818      if (incy .lt. 0) iy = (-n + 1) * incy + 1
819      do 10 i = 1, n
820          dy(iy) = dx(ix)
821          ix = ix + incx
822          iy = iy + incy
823  10 continue
824      return
825  c
826  c          code for both increments equal to 1
827  c
828  c
829  c          clean-up loop
830  c
831      20 m = mod(n, 7)
832      if ( m .eq. 0 ) go to 40
833      do 30 i = 1, m
834          dy(i) = dx(i)
835  30 continue
836      if ( n .lt. 7 ) return
837      40 mp1 = m + 1
838      do 50 i = mp1, n, 7
839          dy(i) = dx(i)
840          dy(i + 1) = dx(i + 1)
841          dy(i + 2) = dx(i + 2)
842          dy(i + 3) = dx(i + 3)
843          dy(i + 4) = dx(i + 4)
844          dy(i + 5) = dx(i + 5)
845          dy(i + 6) = dx(i + 6)
846  50 continue
847      return
848      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, ERMAX, 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: FCVBBDDINIT 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 FCVBBDOP(T, 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 FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, 0.0D0, IER)
222     IF (IER .NE. 0) THEN
223         WRITE(6,92) IER
224 92      FORMAT(///' SUNDIALS_ERROR: FCVBBDREINIT returned IER = ', I5)
225         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
226         STOP
227     ENDIF
228 C
229     CALL FCVSPGMRREINIT(IPRE, IGS, 0.0D0, IER)
230     IF (IER .NE. 0) THEN
231         WRITE(6,93) IER
232 93      FORMAT(///' SUNDIALS_ERROR: FCVSPGMRREINIT returned IER = ', I5)
233         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
234         STOP
235     ENDIF
236 C
237     IF (MYPE .EQ. 0) WRITE(6,95)
238 95      FORMAT(/60('-')///'Preconditioning on right'/)
239     GO TO 40
240 C
241 C     Free the memory and finalize MPI.
242 99      CALL FCVBBDFFREE
243         CALL FCVFFREE
244         CALL FNVFFREEP
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

```

