

Example Programs for CVODE v2.4.0

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

March 24, 2006



UCRL-SM-208110

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: cvdenx	4
2.2	A banded example: cvbanx	6
2.3	A Krylov example: cvkryx	9
3	Parallel example problems	13
3.1	A nonstiff example: cvnonx_p	13
3.2	A user preconditioner example: cvkryx_p	14
3.3	A CVBBDPRE preconditioner example: cvkryx_bbd_p	16
4	Fortran example problems	20
4.1	A serial example: fcvkryx	20
4.2	A parallel example: fcvkryx_bbd_p	21
5	Parallel tests	24
	References	26
A	Listing of cvdenx.c	27
B	Listing of cvbanx.c	34
C	Listing of cvkryx.c	42
D	Listing of cvnonx_p.c	54
E	Listing of cvkryx_p.c	61
F	Listing of cvkryx_bbd_p.c	79
G	Listing of fcvkryx.f	95
H	Listing of fcvkryx_bbd_p.f	112

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):

- **cvdenx** solves a chemical kinetics problem consisting of three rate equations.
This program solves the problem with the BDF method and Newton iteration, with the CVDENSE linear solver and a user-supplied Jacobian routine. It also uses the rootfinding feature of CVODE.
- **cvdenx_uw** is the same as **cvdenx** but demonstrates the user-supplied error weight function feature of CVODE.
- **cvbanx** solves the semi-discrete form of an advection-diffusion equation in 2-D.
This program solves the problem with the BDF method and Newton iteration, with the CVBAND linear solver and a user-supplied Jacobian routine.
- **cvkryx** solves the semi-discrete form of a two-species diurnal kinetics advection-diffusion PDE system in 2-D.
The problem is solved with the BDF/GMRES method (i.e. using the CVSPGMR linear solver) and the block-diagonal part of the Newton matrix as a left preconditioner. A copy of the block-diagonal part of the Jacobian is saved and conditionally reused within the preconditioner setup routine.
- **cvkryx_bp** solves the same problem as **cvkryx**, with the BDF/GMRES method and a banded preconditioner, generated by difference quotients, using the module CVBAND-PRE.
The problem is solved twice: with preconditioning on the left, then on the right.
- **cvkrydem_lin** solves the same problem as **cvkryx**, with the BDF method, but with three Krylov linear solvers: CVSPGMR, CVSPBCG, and CVSPTFQMR.
- **cvdirectdem** is a demonstration program for CVODE with direct linear solvers.
Two separate problems are solved using both the Adams and BDF linear multistep methods in combination with functional and Newton iterations.
The first problem is the Van der Pol oscillator for which the Newton iteration cases use the following types of Jacobian approximations: (1) dense, user-supplied, (2) dense, difference-quotient approximation, (3) diagonal approximation. The second problem is a linear ODE with a banded lower triangular matrix derived from a 2-D advection PDE. In this case, the Newton iteration cases use the following types of Jacobian approximation: (1) banded, user-supplied, (2) banded, difference-quotient approximation, (3) diagonal approximation.
- **cvkrydem_pre** is a demonstration program for CVODE with the Krylov linear solver.
This program solves a stiff ODE system that arises from a system of partial differential

equations. The PDE system is a six-species food web population model, with predator-prey interaction and diffusion on the unit square in two dimensions.

The ODE system is solved using Newton iteration and the CVSPGMR linear solver (scaled preconditioned GMRES).

The preconditioner matrix used is the product of two matrices: (1) a matrix, only defined implicitly, based on a fixed number of Gauss-Seidel iterations using the diffusion terms only; and (2) a block-diagonal matrix based on the partial derivatives of the interaction terms only, using block-grouping.

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

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

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

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

In the following sections, we give detailed descriptions of some (but not all) of these examples. The Appendices contain complete listings of those examples described below. We also give our output files for each of these examples, but users should be cautioned that their results may differ slightly from these. Differences in solution values may differ within the tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

The final section of this report describes a set of tests done with the parallel version of CVODE, using a problem based on the `cvkryx/cvkryx_p` example.

In the descriptions below, we make frequent references to the CVODE User Document [1]. All citations to specific sections (e.g. §5.2) are references to parts of that User Document, unless explicitly stated otherwise.

Note. The examples in the CVODE distribution are written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see §2). As a consequence, they contain portions of code that will not be typically present in a user program. For example, all C example programs make use of the variable `SUNDIALS_EXTENDED_PRECISION` to test if the solver libraries were built in extended precision and use the appropriate conversion specifiers in `printf` functions. Similarly, the FORTRAN examples in FCVODE are automatically pre-processed to generate source code that corresponds to the manner in which the CVODE libraries were built (see §4 in this document for more details).

2 Serial example problems

2.1 A dense example: cvdenx

As an initial illustration of the use of the CVODE package for the integration of IVP ODEs, we give a sample program called `cvdenx.c`. It uses the CVODE dense linear solver module CVDENSE and the NVECTOR_SERIAL module (which provides a serial implementation of NVECTOR) in the solution of a 3-species chemical kinetics problem.

The problem consists of the following three rate equations:

$$\begin{aligned}\dot{y}_1 &= -0.04 \cdot y_1 + 10^4 \cdot y_2 \cdot y_3 \\ \dot{y}_2 &= 0.04 \cdot y_1 - 10^4 \cdot y_2 \cdot y_3 - 3 \cdot 10^7 \cdot y_2^2 \\ \dot{y}_3 &= 3 \cdot 10^7 \cdot y_2^2\end{aligned}\tag{1}$$

on the interval $t \in [0, 4 \cdot 10^{10}]$, with initial conditions $y_1(0) = 1.0$, $y_2(0) = y_3(0) = 0.0$. While integrating the system, we also use the rootfinding feature to find the points at which $y_1 = 10^{-4}$ or at which $y_3 = 0.01$.

For the source, listed in Appendix A, we give a rather detailed explanation of the parts of the program and their interaction with CVODE.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in CVODE header files. The `sundials.types.h` file provides the definition of the type `realtype` (see §5.2 for details). For now, it suffices to read `realtype` as `double`. The `cvode.h` file provides prototypes for the CVODE functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `CVode`. The `cvode_dense.h` file provides the prototype for the `CVDense` function. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. The `sundials_dense.h` file provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements. We have explicitly included `sundials_dense.h`, but this is not necessary because it is included by `cvode_dense.h`.

This program includes two user-defined accessor macros, `Ith` and `IJth` that are useful in writing the problem functions in a form closely matching the mathematical description of the ODE system, i.e. with components numbered from 1 instead of from 0. The `Ith` macro is used to access components of a vector of type `N_Vector` with a serial implementation. It is defined using the NVECTOR_SERIAL accessor macro `NV_Ith_S` which numbers components starting with 0. The `IJth` macro is used to access elements of a dense matrix of type `DenseMat`. It is defined using the DENSE accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macro `NV_Ith_S` is fully described in §7.1. The macro `DENSE_ELEM` is fully described in §5.6.4.

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

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

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

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

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

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

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

The calls to `CVDense` (see §5.5.3) and `CVDenseSetJacFn` (see §5.5.5) specify the CVDENSE linear solver with an analytic Jacobian supplied by the user-supplied function `Jac`.

The actual solution of the ODE initial value problem is accomplished in the loop over values of the output time `tout`. In each pass of the loop, the program calls `CVode` in the `CV_NORMAL` mode, meaning that the integrator is to take steps until it overshoots `tout` and then interpolate to $t = \text{tout}$, 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 §5.5.4 for full details of the call to `CVode`.

Finally, the main program calls `PrintFinalStats` to get and print all of the relevant statistical quantities. It then calls `NV_Destroy` to free the vectors `y` and `abstol`, and `CVodeFree` to free the CVODE memory block.

The function `PrintFinalStats` used here is actually suitable for general use in applications of CVODE to any problem with a dense Jacobian. It calls various `CVodeGet***` and `CVDenseGet***` functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (`nst`), the number of `f` evaluations (`nfe`) (excluding those for difference-quotient Jacobian evaluations), the number of matrix factorizations (`nsetups`), the number of `f` evaluations for Jacobian evaluations (`nfeD = 0` here), the number of Jacobian evaluations (`njeD`), the number of nonlinear (Newton) iterations (`nmi`), the number of nonlinear convergence failures (`ncfn`), the number of local error test failures (`netf`), and the number of `g` (root function) evaluations (`nge`). These optional outputs are described in §5.5.7.

The function `f` is a straightforward expression of the ODEs. It uses the user-defined

macro `Ith` to extract the components of `y` and to load the components of `ydot`. See §5.6.1 for a detailed specification of `f`.

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

The function `Jac` sets the nonzero elements of the Jacobian as a dense matrix. (Zero elements need not be set because `J` is preset to zero.) It uses the user-defined macro `IJth` to reference the elements of a dense matrix of type `DenseMat`. Here the problem size is small, so we need not worry about the inefficiency of using `NV_Ith_S` and `DENSE_ELEM` to access `N_Vector` and `DenseMat` elements. Note that in this example, `Jac` only accesses the `y` and `J` arguments. See §5.6.4 for a detailed description of the dense `Jac` function.

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

```

cvdenx sample output

3-species kinetics problem

At t = 2.6391e-01      y =  9.899653e-01      3.470564e-05      1.000000e-02
  rootsfound[] =    0      1
At t = 4.0000e-01      y =  9.851641e-01      3.386242e-05      1.480205e-02
At t = 4.0000e+00      y =  9.055097e-01      2.240338e-05      9.446793e-02
At t = 4.0000e+01      y =  7.157952e-01      9.183486e-06      2.841956e-01
At t = 4.0000e+02      y =  4.505420e-01      3.222962e-06      5.494548e-01
At t = 4.0000e+03      y =  1.831868e-01      8.941135e-07      8.168123e-01
At t = 4.0000e+04      y =  3.898772e-02      1.621957e-07      9.610121e-01
At t = 4.0000e+05      y =  4.937957e-03      1.984865e-08      9.950620e-01
At t = 4.0000e+06      y =  5.169855e-04      2.069000e-09      9.994830e-01
At t = 2.0798e+07      y =  1.000000e-04      4.000395e-10      9.999000e-01
  rootsfound[] =    1      0
At t = 4.0000e+07      y =  5.199165e-05      2.079772e-10      9.999480e-01
At t = 4.0000e+08      y =  5.212349e-06      2.084950e-11      9.999948e-01
At t = 4.0000e+09      y =  5.193070e-07      2.077229e-12      9.999995e-01
At t = 4.0000e+10      y =  5.829567e-08      2.331827e-13      9.999999e-01

Final Statistics:
nst = 502      nfe = 749      nsetups = 118      nfeLS = 0      nje = 12
nni = 746      ncnf = 0      netf = 30      nge = 532

```

2.2 A banded example: `cvbanx`

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

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

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 $u_{(j-1)+(\text{i}-1)*\text{MY}} = v_{ij}$, so that the ODE system is $\dot{u} = f(u)$, then the system Jacobian $J = \partial f / \partial u$ is a band matrix with upper and lower half-bandwidths both equal to MY . In the example, we take $\text{MX} = 10$ and $\text{MY} = 5$. The source is listed in Appendix B.

The `cvbanx.c` program includes files `cvode_band.h` and `sundials_band.h` in order to use the CVBAND linear solver. The `cvode_band.h` file contains the prototype for the `CVBand` routine. The `sundials_band.h` file contains the definition for band matrix type `BandMat` and the `BAND_COL` and `BAND_COL_ELEM` macros for accessing matrix elements (see §9.2). We have explicitly included `sundials_band.h`, but this is not necessary because it is included by `cvode_band.h`. The file `nvector_serial.h` is included for the definition of the serial `N_Vector` type.

The include lines at the top of the file are followed by definitions of problem constants which include the x and y mesh dimensions, MX and MY , the number of equations NEQ , the scalar absolute tolerance ATOL , the initial time T0 , and the initial output time T1 .

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

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

The `main` program is straightforward. The `CVodeCreate` call specifies the `CV_BDF` method with a `CV_NEWTON` iteration. In the `CVodeMalloc` call, the parameter `SS` indicates scalar relative and absolute tolerances, and pointers `&reltol` and `&abstol` to these values are passed. The call to `CVBand` (see §5.5.3) specifies the CVBAND linear solver, and specifies that both half-bandwidths of the Jacobian are equal to MY . The call to `CVBandSetJacFn` (see §5.5.5) specifies that a user-supplied Jacobian function `Jac` is to be used and that a pointer to `data` should be passed to `Jac` every time it is called. The actual solution of the problem is performed by the call to `CVode` within the loop over the output times `tout`. The max-norm of the solution vector (from a call to `N_VMaxNorm`) and the cumulative number of time steps (from a call to `CVodeGetNumSteps`) are printed at each output time. Finally, the calls to `PrintFinalStats`, `N_VDestroy`, and `CVodeFree` print statistics and free problem memory.

Following the `main` program in the `cvbanx.c` file are definitions of five functions: `f`, `Jac`, `SetIC`, `PrintFinalStats`, and `check_flag`. The last three functions are called only from within the `cvbanx.c` file. The `SetIC` function sets the initial dependent variable vector; `PrintFinalStats` gets and prints statistics at the end of the run; and `check_flag` aids in checking return values. The statistics printed include counters such as the total number of steps (`nst`), `f` evaluations (excluding those for 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 §5.5.7. Note that `PrintFinalStats` is suitable for general use in applications of CVODE to any problem with a banded Jacobian.

The `f` function implements the central difference approximation (3) with u identically zero on the boundary. The constant coefficients $(\Delta x)^{-2}$, $.5(2\Delta x)^{-1}$, and $(\Delta y)^{-2}$ are computed only once at the beginning of `main`, and stored in the locations `data->hdcoef`, `data->hacoef`, and `data->vdcoef`, respectively. When `f` receives the `data` pointer (renamed `f_data` here), it pulls out these values from storage in the local variables `hordc`, `horac`, and `verdc`. It then uses these to construct the diffusion and advection terms, which are combined to form `udot`. Note the extra lines setting out-of-bounds values of u to zero.

The `Jac` function is an expression of the derivatives

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

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 != 1)` etc. in front of the calls to `BAND_COL_ELEM` prevent illegal indexing. See §5.6.5 for a detailed description of the banded `Jac` function.

The output generated by `cvbanx` is shown below.

```

----- cvbanx sample output -----

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

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

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

```

2.3 A Krylov example: cvkryx

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

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

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

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 `cvode_spgmr.h` and `sundials_math.h`. The first contains constants and function prototypes associated with the SPGMR method, including the values of the `pretype` argument to `CVSpgmr`. The inclusion of `sundials_math.h` is done to access the `SQR` macro for the square of a `realtype` number.

The main program calls `CVodeCreate` specifying the `CV_BDF` method and `CV_NEWTON` iteration, and then calls `CVodeMalloc` with scalar tolerances. It calls `CVSpgmr` (see §5.5.3) to specify the CVSPGMR linear solver with left preconditioning, and the default value (indicated by a zero argument) for `maxl`. The Gram-Schmidt orthogonalization is set to `MODIFIED_GS` through the function `CVSpgmrSetGSType`. Next, user-supplied preconditioner setup and solve functions, `Precond` and `PSolve`, as well as the `data` pointer passed to `Precond` and `PSolve` whenever these are called, See §5.5.5 for details on the `CVSpgmrSetPreconditioner` function.

Then for a sequence of `tout` values, `CVode` is called in the `CV_NORMAL` mode, sampled output is printed, and the return value is tested for error conditions. After that, `PrintFinalStats`

is called to get and print final statistics, and memory is freed by calls to `N_VDestroy`, `FreeUserData`, and `CVodeFree`. The printed statistics include various counters, such as the total numbers of steps (`nst`), of `f` evaluations (excluding those for Jv product evaluations) (`nfe`), of `f` evaluations for Jv evaluations (`nfel`), of nonlinear iterations (`nni`), of linear (Krylov) iterations (`nli`), of preconditioner setups (`nsetups`), of preconditioner evaluations (`npe`), and of preconditioner solves (`nps`), among others. Also printed are the lengths of the problem-dependent real and integer workspaces used by the main integrator `CVode`, denoted `lenrw` and `leniw`, and those used by `CVSPGMR`, denoted `llrw` and `lliw`. All of these optional outputs are described in §5.5.7. The `PrintFinalStats` function is suitable for general use in applications of `CVODE` to any problem with the `SPGMR` linear solver.

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

The preconditioner used here is the block-diagonal part of the true Newton matrix. It is generated and factored in the `Precond` routine (see §5.6.8) and backsolved in the `PSolve` routine (see §5.6.7). Its diagonal blocks are 2×2 matrices that include the interaction Jacobian elements and the diagonal contribution of the diffusion Jacobian elements. The block-diagonal part of the Jacobian itself, J_{bd} , is saved in separate storage each time it is generated, on calls to `Precond` with `jok == FALSE`. On calls with `jok == TRUE`, signifying that saved Jacobian data can be reused, the preconditioner $P = I - \gamma J_{bd}$ is formed from the saved matrix J_{bd} and factored. (A call to `Precond` with `jok == TRUE` can only occur after a prior call with `jok == FALSE`.) The `Precond` routine must also set the value of `jcur`, i.e. `*jcurPtr`, to `TRUE` when J_{bd} is re-evaluated, and `FALSE` otherwise, to inform `CVSPGMR` of the status of Jacobian data.

We need to take a brief detour to explain one last important aspect of the `cvkryx.c` program. The generic `DENSE` solver contains two sets of functions: one for “large” matrices and one for “small” matrices. The large dense functions work with the type `DenseMat`, while the small dense functions work with `realtype **` as the underlying dense matrix types. The `CVDENSE` linear solver uses the type `DenseMat` for the $N \times N$ dense Jacobian and Newton matrices, and calls the large matrix functions. But to avoid the extra layer of function calls, `cvkryx.c` uses the small dense functions for all operations on the 2×2 preconditioner blocks. Thus it includes `sundials_smalldense.h`, and calls the small dense matrix functions `denalloc`, `dencopy`, `denscale`, `denaddI`, `denfree`, `denfreepiv`, `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 §9.1.

In addition to the functions called by `CVODE`, `cvkryx.c` includes definitions of several private functions. These are: `AllocUserData` to allocate space for J_{bd} , P , and the pivot arrays; `InitUserData` to load problem constants in the `data` block; `FreeUserData` to free that block; `SetInitialProfiles` to load the initial values in `y`; `PrintOutput` to 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 `cvkryx.c` is shown below. Note that the number of precondi-

tioner evaluations, `npe`, is much smaller than the number of preconditioner setups, `nsetups`, as a result of the Jacobian re-use scheme.

cvkryx sample output

2-species diurnal advection-diffusion problem

t = 7.20e+03	no. steps = 219	order = 5	stepsize = 1.59e+02	
c1 (bot.left/middle/top rt.) =	1.047e+04	2.964e+04	1.119e+04	
c2 (bot.left/middle/top rt.) =	2.527e+11	7.154e+11	2.700e+11	
t = 1.44e+04	no. steps = 251	order = 5	stepsize = 3.77e+02	
c1 (bot.left/middle/top rt.) =	6.659e+06	5.316e+06	7.301e+06	
c2 (bot.left/middle/top rt.) =	2.582e+11	2.057e+11	2.833e+11	
t = 2.16e+04	no. steps = 277	order = 5	stepsize = 2.75e+02	
c1 (bot.left/middle/top rt.) =	2.665e+07	1.036e+07	2.931e+07	
c2 (bot.left/middle/top rt.) =	2.993e+11	1.028e+11	3.313e+11	
t = 2.88e+04	no. steps = 306	order = 4	stepsize = 2.05e+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 = 347	order = 4	stepsize = 8.74e+01	
c1 (bot.left/middle/top rt.) =	1.404e+04	2.029e+04	1.561e+04	
c2 (bot.left/middle/top rt.) =	3.387e+11	4.894e+11	3.765e+11	
t = 4.32e+04	no. steps = 411	order = 4	stepsize = 4.51e+02	
c1 (bot.left/middle/top rt.) =	1.018e-07	-5.353e-07	1.171e-07	
c2 (bot.left/middle/top rt.) =	3.382e+11	1.355e+11	3.804e+11	
t = 5.04e+04	no. steps = 429	order = 4	stepsize = 3.42e+02	
c1 (bot.left/middle/top rt.) =	1.529e-12	2.028e-12	1.738e-12	
c2 (bot.left/middle/top rt.) =	3.358e+11	4.930e+11	3.864e+11	
t = 5.76e+04	no. steps = 442	order = 5	stepsize = 4.25e+02	
c1 (bot.left/middle/top rt.) =	3.314e-13	3.343e-12	3.423e-13	
c2 (bot.left/middle/top rt.) =	3.320e+11	9.650e+11	3.909e+11	
t = 6.48e+04	no. steps = 452	order = 5	stepsize = 7.84e+02	
c1 (bot.left/middle/top rt.) =	-2.730e-13	-2.106e-12	-2.869e-13	
c2 (bot.left/middle/top rt.) =	3.313e+11	8.922e+11	3.963e+11	
t = 7.20e+04	no. steps = 461	order = 5	stepsize = 7.84e+02	
c1 (bot.left/middle/top rt.) =	-6.822e-14	-2.240e-12	-2.888e-14	
c2 (bot.left/middle/top rt.) =	3.330e+11	6.186e+11	4.039e+11	
t = 7.92e+04	no. steps = 470	order = 5	stepsize = 7.84e+02	
c1 (bot.left/middle/top rt.) =	9.495e-16	5.968e-14	-3.437e-16	
c2 (bot.left/middle/top rt.) =	3.334e+11	6.669e+11	4.120e+11	
t = 8.64e+04	no. steps = 479	order = 5	stepsize = 7.84e+02	
c1 (bot.left/middle/top rt.) =	5.210e-16	1.513e-14	2.143e-16	
c2 (bot.left/middle/top rt.) =	3.352e+11	9.107e+11	4.163e+11	

Final Statistics..

lenrw	=	2089	leniw	=	50
lenrwLS	=	2046	leniwLS	=	10
nst	=	479			
nfe	=	610	nfeLS	=	627
nni	=	607	nli	=	627
nsetups	=	75	netf	=	25
npe	=	9	nps	=	1171
ncfn	=	0	ncfl	=	0

3 Parallel example problems

3.1 A nonstiff example: `cvnonx_p`

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

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

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

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

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

In the parallel processing environment, we may think of the several processors as being laid out on a straight line with each processor to compute its contiguous subset of the solution vector. Consequently the computation of the right hand side of Eq. (9) requires that each interior processor must pass the first component of its block of the solution vector to its left-hand neighbor, acquire the last component of that neighbor's block, pass the last component of its block of the solution vector to its right-hand neighbor, and acquire the first component of that neighbor's block. If the processor is the first (0th) or last processor, then communication to the left or right (respectively) is not required.

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

The `cvnonx_p.c` file begins with `#include` lines, including lines for `nvector_parallel` to access the parallel `N_Vector` type and related macros, and for `mpi.h` to access MPI types and constants. Following that are definitions of problem constants and a data block for communication with the `f` routine. That block includes the number of PEs, the index of the local PE, and the MPI communicator.

The `main` program begins with MPI calls to initialize MPI and to set multi-processor environment parameters `npes` (number of PEs) and `my_pe` (local PE index). The local vector length is set according to `npes` and the problem size `NEQ` (which may or may not be multiple of `npes`). The value `my_base` is the base value for computing global indices (from 1 to `NEQ`) for the local vectors. The solution vector `u` is created with a call to `N_VNew_Parallel` and loaded with a call to `SetIC`. The calls to `CVodeCreate` and `CVodeMalloc` specify a CVODE solution with the nonstiff method and scalar tolerances. The call to `CVodeSetFdata` insures that the pointer `data` is passed to the `f` routine whenever it is called. A heading is printed (if on processor 0). In a loop over `tout` values, `CVode` is called, and the return value checked for

errors. The max-norm of the solution and the total number of time steps so far are printed at each output point. Finally, some statistical counters are printed, memory is freed, and MPI is finalized.

The `SetIC` routine uses the last two arguments passed to it to compute the set of global indices (`my_base+1` to `my_base+my_length`) corresponding to the local part of the solution vector `u`, and then to load the corresponding initial values. The `PrintFinalStats` routine uses `CVodeGet***` calls to get various counters, and then prints these. The counters are: `nst` (number of steps), `nfe` (number of `f` evaluations), `nni` (number of nonlinear iterations), `netf` (number of error test failures), and `ncfn` (number of nonlinear convergence failures). This routine is suitable for general use with CVODE applications to nonstiff problems.

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

The `cvnonx.p.c` file includes a routine `check_flag` that checks the return values from calls in `main`. This routine was written to be used by any parallel SUNDIALS application.

The output below is for `cvnonx.p` with `MX = 10` and four processors. Varying the number of processors will alter the output, only because of roundoff-level differences in various vector operations. The fairly high value of `ncfn` indicates that this problem is on the borderline of being stiff.

```

----- cvnonx.p sample output -----
1-D advection-diffusion equation, mesh size = 10

Number of PEs =    4

At t = 0.00  max.norm(u) = 1.569909e+01  nst =    0
At t = 0.50  max.norm(u) = 3.052881e+00  nst =  113
At t = 1.00  max.norm(u) = 8.753188e-01  nst =  191
At t = 1.50  max.norm(u) = 2.494926e-01  nst =  265
At t = 2.00  max.norm(u) = 7.109707e-02  nst =  339
At t = 2.50  max.norm(u) = 2.026223e-02  nst =  418
At t = 3.00  max.norm(u) = 5.770281e-03  nst =  492
At t = 3.50  max.norm(u) = 1.647117e-03  nst =  556
At t = 4.00  max.norm(u) = 4.714592e-04  nst =  645
At t = 4.50  max.norm(u) = 1.365028e-04  nst =  720
At t = 5.00  max.norm(u) = 3.878829e-05  nst =  794

Final Statistics:

nst = 794      nfe = 1400      nni = 1396      ncfn = 117      netf = 5

```

3.2 A user preconditioner example: `cvkryx.p`

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

differencing, to obtain an ODE system $\dot{u} = f(t, u)$ representing (4). But in this case, the discrete solution vector is distributed over many processors. Specifically, we may think of the processors as being laid out in a rectangle, and each processor being assigned a subgrid of size $\text{MXSUB} \times \text{MYSUB}$ 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 $\text{MX} \times \text{MY}$ with $\text{MX} = \text{NPEX} \times \text{MXSUB}$ and $\text{MY} = \text{NPEY} \times \text{MYSUB}$, and the size of the ODE system is $2 \cdot \text{MX} \cdot \text{MY}$.

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

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

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

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

```

----- cvkryx.p sample output -----

2-species diurnal advection-diffusion problem

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

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

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

```

```

t = 2.88e+04    no. steps = 301    order = 5    stepsize = 3.46e+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 = 329    order = 5    stepsize = 1.30e+02
At bottom left: c1, c2 =      1.404e+04    3.387e+11
At top right:   c1, c2 =      1.561e+04    3.765e+11

t = 4.32e+04    no. steps = 380    order = 4    stepsize = 3.91e+02
At bottom left: c1, c2 =      3.842e-07    3.382e+11
At top right:   c1, c2 =      4.326e-07    3.804e+11

t = 5.04e+04    no. steps = 394    order = 5    stepsize = 4.05e+02
At bottom left: c1, c2 =     -4.432e-09    3.358e+11
At top right:   c1, c2 =     -1.691e-08    3.864e+11

t = 5.76e+04    no. steps = 410    order = 5    stepsize = 2.20e+02
At bottom left: c1, c2 =     -2.084e-07    3.320e+11
At top right:   c1, c2 =     -7.777e-08    3.909e+11

t = 6.48e+04    no. steps = 430    order = 5    stepsize = 5.82e+02
At bottom left: c1, c2 =     -9.528e-11    3.313e+11
At top right:   c1, c2 =     -3.991e-11    3.963e+11

t = 7.20e+04    no. steps = 440    order = 5    stepsize = 7.15e+02
At bottom left: c1, c2 =     -1.425e-11    3.330e+11
At top right:   c1, c2 =     -5.963e-12    4.039e+11

t = 7.92e+04    no. steps = 450    order = 5    stepsize = 7.15e+02
At bottom left: c1, c2 =      1.503e-12    3.334e+11
At top right:   c1, c2 =      6.284e-13    4.120e+11

t = 8.64e+04    no. steps = 460    order = 5    stepsize = 7.15e+02
At bottom left: c1, c2 =     -8.982e-14    3.352e+11
At top right:   c1, c2 =     -3.754e-14    4.163e+11

```

Final Statistics:

```

lenrw  = 2089    leniw  = 120
lenrwls = 2046    leniwls = 80
nst     = 460
nfe     = 599    nfels  = 613
nni     = 595    nli    = 613
nsetups = 80     netf   = 29
npe     = 8      nps    = 1153
ncfn    = 0      ncfl   = 0

```

3.3 A CVBBDPRE preconditioner example: cvkryx_bbd_p

In this example, `cvkryx_bbd_p`, we solve the same problem in `cvkryx_p` above, but instead of supplying the preconditioner, we use the CVBBDPRE module, which generates and uses a band-block-diagonal preconditioner. The half-bandwidths of the Jacobian block on each processor are both equal to `2*MXSUB`, and that is the value supplied as `mudq` and `mldq` in the call to `CVBBDPrecAlloc`. But in order to reduce storage and computation costs for preconditioning, we supply the values `mukeep = mlkeep = 2 (= NVARs)` as the half-bandwidths of the retained

band matrix blocks. This means that the Jacobian elements are computed with a difference quotient scheme using the true bandwidth of the block, but only a narrow band matrix (bandwidth 5) is kept as the preconditioner. The source is listed in Appendix F.

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

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

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

```

cvkryx_bbd.p sample output

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

Preconditioner type is:  jpre = PREC_LEFT

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

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

t = 2.16e+04   no. steps = 247   order = 5   stepsize = 3.01e+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 = 8.09e+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 =     -2.111e-07    3.382e+11
At top right:    c1, c2 =      3.512e-08    3.804e+11

t = 5.04e+04   no. steps = 399   order = 5   stepsize = 3.90e+02
At bottom left:  c1, c2 =      1.219e-08    3.358e+11
At top right:    c1, c2 =     -1.339e-07    3.864e+11

```

t = 5.76e+04 no. steps = 412 order = 5 stepsize = 4.44e+02
 At bottom left: c1, c2 = -6.333e-11 3.320e+11
 At top right: c1, c2 = 7.610e-10 3.909e+11

t = 6.48e+04 no. steps = 431 order = 4 stepsize = 5.22e+02
 At bottom left: c1, c2 = 9.825e-11 3.313e+11
 At top right: c1, c2 = -1.284e-09 3.963e+11

t = 7.20e+04 no. steps = 444 order = 4 stepsize = 5.22e+02
 At bottom left: c1, c2 = -3.134e-15 3.330e+11
 At top right: c1, c2 = 4.376e-14 4.039e+11

t = 7.92e+04 no. steps = 458 order = 4 stepsize = 5.22e+02
 At bottom left: c1, c2 = -3.057e-19 3.334e+11
 At top right: c1, c2 = 5.128e-18 4.120e+11

t = 8.64e+04 no. steps = 472 order = 4 stepsize = 5.22e+02
 At bottom left: c1, c2 = -2.733e-21 3.352e+11
 At top right: c1, c2 = 2.359e-19 4.163e+11

Final Statistics:

lenrw	=	2089	leniw	=	120
lenrwls	=	2046	leniwls	=	80
nst	=	472			
nfe	=	626	nfels	=	594
nni	=	622	nli	=	594
nsetups	=	89	netf	=	35
npe	=	9	nps	=	1167
ncfn	=	0	ncfl	=	2

In CVBBDPRE: real/integer local work space sizes = 600, 50
 no. fllocal 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 = 308 order = 4 stepsize = 2.16e+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 = 348 order = 4 stepsize = 5.82e+01

```

At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04    no. steps = 405    order = 4    stepsize = 3.94e+02
At bottom left:  c1, c2 =   -3.403e-10    3.382e+11
At top right:    c1, c2 =   -3.639e-10    3.804e+11

t = 5.04e+04    no. steps = 418    order = 5    stepsize = 6.83e+02
At bottom left:  c1, c2 =   -8.094e-11    3.358e+11
At top right:    c1, c2 =   -6.552e-11    3.864e+11

t = 5.76e+04    no. steps = 429    order = 5    stepsize = 3.86e+02
At bottom left:  c1, c2 =    3.492e-12    3.320e+11
At top right:    c1, c2 =   -2.662e-12    3.909e+11

t = 6.48e+04    no. steps = 442    order = 5    stepsize = 5.88e+02
At bottom left:  c1, c2 =   -7.105e-12    3.313e+11
At top right:    c1, c2 =   -9.102e-12    3.963e+11

t = 7.20e+04    no. steps = 454    order = 5    stepsize = 5.88e+02
At bottom left:  c1, c2 =   -2.880e-13    3.330e+11
At top right:    c1, c2 =    4.282e-14    4.039e+11

t = 7.92e+04    no. steps = 467    order = 5    stepsize = 5.88e+02
At bottom left:  c1, c2 =    5.693e-14    3.334e+11
At top right:    c1, c2 =   -2.980e-15    4.120e+11

t = 8.64e+04    no. steps = 479    order = 5    stepsize = 5.88e+02
At bottom left:  c1, c2 =    3.034e-15    3.352e+11
At top right:    c1, c2 =    1.371e-16    4.163e+11

```

Final Statistics:

```

lenrw   = 2089    leniw   = 120
lenrwls = 2046    leniwls = 80
nst      = 479
nfe      = 636    nfels   = 807
nni      = 632    nli     = 807
nsetups  = 107    netf    = 37
npe      = 9      nps     = 1322
ncfn     = 0      ncfl    = 0

```

```

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

```

4 Fortran example problems

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

4.1 A serial example: `fcvkryx`

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

The main program begins with a call to `INITKX`, which sets problem parameters, loads these into arrays `IPAR` and `RPAR` for use by other routines, and loads `Y` with its initial values. It calls `FNVINITS`, `FCVMALLOC`, `FCVSPGMR`, `FCVSPGMRSETPSET`, and `FCVSPGMRSETPSOL` to initialize the `NVECTOR_SERIAL` module, the main solver memory, and the `CVSPGMR` module, and to specify user-supplied preconditioner setup and solve routines. It calls `FCVODE` in a loop over `TOUT` values, with printing of selected solution values and performance data (from the `IOUT` and `ROUT` arrays). At the end, it prints a number of performance counters, and frees memory with calls to `FCVFREE`.

In `fcvkryx.f`, the `FCVFUN` routine is a 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 `fcvkryx.f` consists of routines from LINPACK and the BLAS needed for matrix and vector operations.

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

fcvkryx sample output									
Krylov example problem:									
Kinetics-transport, NEQ = 200									
t =	0.720E+04	nst =	219	q =	5	h =	0.158696E+03		
c1 (bot.left/middle/top rt.) =	0.104683E+05	0.296373E+05	0.111853E+05						
c2 (bot.left/middle/top rt.) =	0.252672E+12	0.715376E+12	0.269977E+12						
t =	0.144E+05	nst =	251	q =	5	h =	0.377205E+03		
c1 (bot.left/middle/top rt.) =	0.665902E+07	0.531602E+07	0.730081E+07						
c2 (bot.left/middle/top rt.) =	0.258192E+12	0.205680E+12	0.283286E+12						
t =	0.216E+05	nst =	277	q =	5	h =	0.274582E+03		
c1 (bot.left/middle/top rt.) =	0.266498E+08	0.103636E+08	0.293077E+08						


```

c2 (bot.left/middle/top rt.) = 0.299279E+12 0.102810E+12 0.331344E+12

t = 0.288E+05 nst = 301 q = 5 h = 0.221437E+03
c1 (bot.left/middle/top rt.) = 0.870209E+07 0.129197E+08 0.965002E+07
c2 (bot.left/middle/top rt.) = 0.338035E+12 0.502929E+12 0.375096E+12

t = 0.360E+05 nst = 347 q = 4 h = 0.464845E+02
c1 (bot.left/middle/top rt.) = 0.140404E+05 0.202903E+05 0.156090E+05
c2 (bot.left/middle/top rt.) = 0.338677E+12 0.489443E+12 0.376516E+12

t = 0.432E+05 nst = 404 q = 4 h = 0.385936E+03
c1 (bot.left/middle/top rt.) = 0.660556E-08 -0.335915E-04 0.208242E-06
c2 (bot.left/middle/top rt.) = 0.338233E+12 0.135489E+12 0.380352E+12

t = 0.504E+05 nst = 418 q = 5 h = 0.420821E+03
c1 (bot.left/middle/top rt.) = 0.381331E-07 -0.380748E-04 0.210575E-06
c2 (bot.left/middle/top rt.) = 0.335816E+12 0.493024E+12 0.386445E+12

t = 0.576E+05 nst = 430 q = 5 h = 0.496448E+03
c1 (bot.left/middle/top rt.) = -0.326413E-08 0.359853E-05 -0.169658E-07
c2 (bot.left/middle/top rt.) = 0.332031E+12 0.964982E+12 0.390900E+12

t = 0.648E+05 nst = 442 q = 5 h = 0.759640E+03
c1 (bot.left/middle/top rt.) = 0.201505E-08 -0.222511E-05 0.104495E-07
c2 (bot.left/middle/top rt.) = 0.331303E+12 0.892188E+12 0.396342E+12

t = 0.720E+05 nst = 452 q = 5 h = 0.759640E+03
c1 (bot.left/middle/top rt.) = -0.331242E-09 0.367385E-06 -0.171960E-08
c2 (bot.left/middle/top rt.) = 0.332972E+12 0.618620E+12 0.403885E+12

t = 0.792E+05 nst = 461 q = 5 h = 0.759640E+03
c1 (bot.left/middle/top rt.) = 0.120687E-10 -0.134246E-07 0.626981E-10
c2 (bot.left/middle/top rt.) = 0.333441E+12 0.666909E+12 0.412026E+12

t = 0.864E+05 nst = 471 q = 5 h = 0.759640E+03
c1 (bot.left/middle/top rt.) = -0.563851E-12 0.630946E-09 -0.293343E-11
c2 (bot.left/middle/top rt.) = 0.335178E+12 0.910633E+12 0.416251E+12

```

Final statistics:

```

number of steps      = 471      number of f evals.      = 627
number of prec. setups = 82
number of prec. evals. = 8      number of prec. solves = 1185
number of nonl. iters. = 623    number of lin. iters.  = 623
average Krylov subspace dimension (NLI/NNI) = 0.100000E+01
number of conv. failures.. nonlinear = 0 linear = 0
number of error test failures = 32

```

4.2 A parallel example: fcvkryx_bbd_p

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

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

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, `fcvkryx_bbd_p.f`, is listed in Appendix H. It begins with MPI calls to initialize MPI and to get the number of processors and local processor index. The linear solver specification is done with calls to `FCVBBDINIT` and `FCVBBDSPGMR`. In a loop over TOUT values, it calls `FCVODE` and prints the step and f evaluation counters. After that, it computes and prints the maximum global error, and all the relevant performance counters. Those specific to CVBBDPRE are obtained by a call to `FCVBBDOPT`. To prepare for the second run, the program calls `FCVREINIT`, `FCVBBDREINIT`, and `FCVSPGMRREINIT`, in addition to resetting the initial conditions. Finally, it frees memory and terminates MPI. Notice that in the `FCVFUN` routine, the local processor index `MYPE` and the local vector size `NLOCAL` are used to form the global index values needed to evaluate the right-hand side of Eq. (10).

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

```

fcvkryx_bbd_p sample output

Diagonal test problem:

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

Preconditioning on left

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

Max. absolute error is 0.28E-08

Final statistics:

number of steps          = 359      number of f evals.      = 406
number of prec. setups   = 38
number of prec. evals.   = 7        number of prec. solves   = 728
number of nonl. iters.   = 402      number of lin. iters.   = 364
average Krylov subspace dimension (NLI/NNI) = 0.9055
number of conv. failures.. nonlinear = 0 linear = 0
number of error test failures = 5

```

```

main solver real/int workspace sizes = 489 120
linear solver real/int workspace sizes = 446 80
In CVBBDPRE:
real/int local workspace = 20 10
number of g evals. = 14

```

Preconditioning on right

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

Max. absolute error is 0.28E-08

Final statistics:

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

```

In CVBBDPRE:
real/int local workspace = 20 10
number of g evals. = 14

```

5 Parallel tests

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

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

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

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, `maxl`, to 10 (from its default value of 5).

For each of the three test cases, the test program was run on a Cray-T3D (256 processors) with each of three different message-passing libraries:

- `MPICH`: an implementation of MPI on top of the Chameleon library
- `EPCC`: an implementation of MPI by the Edinburgh Parallel Computer Centre
- `SHMEM`: Cray's Shared Memory Library

The following table gives the run time and selected performance counters for these 9 runs. In all cases, the solutions agreed well with each other, showing expected small variations with grid size. In the table, M-P denotes the message-passing library, `RT` is the reported run time in CPU seconds, `nst` is the number of time steps, `nfe` is the number of f evaluations, `nni` is the number of nonlinear (Newton) iterations, `nli` is the number of linear (Krylov) iterations, and `npe` is the number of evaluations of the preconditioner.

Some of the results were as expected, and some were surprising. For a given mesh size, variations in performance counts were small or absent, except for moderate (but still acceptable) variations for `SHMEM` in the smallest case. The increase in costs with mesh size can be attributed to a decline in the quality of the preconditioner, which neglects most of the spatial coupling. The preconditioner quality can be inferred from the ratio `nli/nni`, which is the average number of Krylov iterations per Newton iteration. The most interesting (and unexpected) result is the variation of run time with library: `SHMEM` is the most efficient,

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

Table 1: Parallel CVODE test results vs problem size and message-passing library

but EPCC is a very close second, and MPICH loses considerable efficiency by comparison, as the problem size grows. This means that the highly portable MPI version of CVODE, with an appropriate choice of MPI implementation, is fully competitive with the Cray-specific version using the SHMEM library. While the overall costs do not prepresent a well-scaled parallel algorithm (because of the preconditioner choice), the cost per function evaluation is quite flat for EPCC and SHMEM, at .033 to .037 (for MPICH it ranges from .044 to .068).

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

References

- [1] A. C. Hindmarsh and R. Serban. User Documentation for CVODE v2.4.0. Technical Report UCRL-SM-208108, LLNL, 2005.
- [2] M. R. Wittman. Testing of PVODE, a Parallel ODE Solver. Technical Report UCRL-ID-125562, LLNL, August 1996.

A Listing of cvdenx.c

```
1  /*
2  * -----
3  * $Revision: 1.4 $
4  * $Date: 2006/03/17 16:57:54 $
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 cvdenx.c */
34
35 #include "cvsode.h" /* prototypes for CVODE fcts. and consts. */
36 #include "nvector_serial.h" /* serial N_Vector types, fcts., and macros */
37 #include "cvsode_dense.h" /* prototype for CVDense */
38 #include "sundials_dense.h" /* definitions DenseMat DENSE_ELEM */
39 #include "sundials_types.h" /* definition of type realtype */
40
41 /* User-defined vector and matrix accessor macros: Ith, IJth */
42
43 /* These macros are defined in order to write code which exactly matches
44 the mathematical problem description given above.
45
46 Ith(v,i) references the ith component of the vector v, where i is in
47 the range [1..NEQ] and NEQ is defined below. The Ith macro is defined
48 using the NV_Ith macro in nvector.h. NV_Ith numbers the components of
49 a vector starting from 0.
50
51 IJth(A,i,j) references the (i,j)th element of the dense matrix A, where
52 i and j are in the range [1..NEQ]. The IJth macro is defined using the
53 DENSE_ELEM macro in dense.h. DENSE_ELEM numbers rows and columns of a
54 dense matrix starting from 0. */
55
56 #define Ith(v,i) NV_Ith_S(v,i-1) /* Ith numbers components 1..NEQ */
57 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1) /* IJth numbers rows,cols 1..NEQ */
```

```

58
59
60 /* Problem Constants */
61
62 #define NEQ      3                /* number of equations */
63 #define Y1       RCONST(1.0)     /* initial y components */
64 #define Y2       RCONST(0.0)
65 #define Y3       RCONST(0.0)
66 #define RTOL     RCONST(1.0e-4)  /* scalar relative tolerance */
67 #define ATOL1    RCONST(1.0e-8)  /* vector absolute tolerance components */
68 #define ATOL2    RCONST(1.0e-14)
69 #define ATOL3    RCONST(1.0e-6)
70 #define T0       RCONST(0.0)     /* initial time */
71 #define T1       RCONST(0.4)     /* first output time */
72 #define TMULT    RCONST(10.0)    /* output time factor */
73 #define NOUT     12              /* number of output times */
74
75
76 /* Functions Called by the Solver */
77
78 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data);
79
80 static int g(realtype t, N_Vector y, realtype *gout, void *g_data);
81
82 static int Jac(long int N, DenseMat J, realtype t,
83               N_Vector y, N_Vector fy, void *jac_data,
84               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
85
86 /* Private functions to output results */
87
88 static void PrintOutput(realtype t, realtype y1, realtype y2, realtype y3);
89 static void PrintRootInfo(int root_f1, int root_f2);
90
91 /* Private function to print final statistics */
92
93 static void PrintFinalStats(void *cvmem);
94
95 /* Private function to check function return values */
96
97 static int check_flag(void *flagvalue, char *funcname, int opt);
98
99
100 /*
101  *-----
102  * Main Program
103  *-----
104  */
105
106 int main()
107 {
108     realtype reltol, t, tout;
109     N_Vector y, abstol;
110     void *cvmem;
111     int flag, flagr, iout;
112     int rootsfound[2];
113
114     y = abstol = NULL;
115     cvmem = NULL;
116

```



```

117  /* Create serial vector of length NEQ for I.C. and abstol */
118  y = N_VNew_Serial(NEQ);
119  if (check_flag((void *)y, "N_VNew_Serial", 0)) return(1);
120  abstol = N_VNew_Serial(NEQ);
121  if (check_flag((void *)abstol, "N_VNew_Serial", 0)) return(1);
122
123  /* Initialize y */
124  Ith(y,1) = Y1;
125  Ith(y,2) = Y2;
126  Ith(y,3) = Y3;
127
128  /* Set the scalar relative tolerance */
129  reltol = RTOL;
130  /* Set the vector absolute tolerance */
131  Ith(abstol,1) = ATOL1;
132  Ith(abstol,2) = ATOL2;
133  Ith(abstol,3) = ATOL3;
134
135  /*
136   Call CVodeCreate to create the solver memory:
137
138   CV_BDF      specifies the Backward Differentiation Formula
139   CV_NEWTON   specifies a Newton iteration
140
141   A pointer to the integrator problem memory is returned and stored in ccode_mem.
142  */
143
144  ccode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
145  if (check_flag((void *)ccode_mem, "CVodeCreate", 0)) return(1);
146
147  /*
148   Call CVodeMalloc to initialize the integrator memory:
149
150   ccode_mem is the pointer to the integrator memory returned by CVodeCreate
151   f          is the user's right hand side function in y'=f(t,y)
152   T0         is the initial time
153   y          is the initial dependent variable vector
154   CV_SV      specifies scalar relative and vector absolute tolerances
155   &reltol    is a pointer to the scalar relative tolerance
156   abstol     is the absolute tolerance vector
157  */
158
159  flag = CVodeMalloc(ccode_mem, f, T0, y, CV_SV, reltol, abstol);
160  if (check_flag(&flag, "CVodeMalloc", 1)) return(1);
161
162  /* Call CVodeRootInit to specify the root function g with 2 components */
163  flag = CVodeRootInit(ccode_mem, 2, g, NULL);
164  if (check_flag(&flag, "CVodeRootInit", 1)) return(1);
165
166  /* Call CVDense to specify the CVDENSE dense linear solver */
167  flag = CVDense(ccode_mem, NEQ);
168  if (check_flag(&flag, "CVDense", 1)) return(1);
169
170  /* Set the Jacobian routine to Jac (user-supplied) */
171  flag = CVDenseSetJacFn(ccode_mem, Jac, NULL);
172  if (check_flag(&flag, "CVDenseSetJacFn", 1)) return(1);
173
174  /* In loop, call CVode, print results, and test for error.
175   Break out of loop when NOUT preset output times have been reached.  */

```

```

176     printf("\n3-species_kinetics_problem\n\n");
177
178     iout = 0; tout = T1;
179     while(1) {
180         flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
181         PrintOutput(t, Ith(y,1), Ith(y,2), Ith(y,3));
182
183         if (flag == CV_ROOT_RETURN) {
184             flagr = CVodeGetRootInfo(cvode_mem, rootsfound);
185             if (check_flag(&flagr, "CVodeGetRootInfo", 1)) return(1);
186             PrintRootInfo(rootsfound[0], rootsfound[1]);
187         }
188
189         if (check_flag(&flag, "CVode", 1)) break;
190         if (flag == CV_SUCCESS) {
191             iout++;
192             tout *= TMULT;
193         }
194
195         if (iout == NOUT) break;
196     }
197
198     /* Print some final statistics */
199     PrintFinalStats(cvode_mem);
200
201     /* Free y vector */
202     N_VDestroy_Serial(y);
203
204     /* Free integrator memory */
205     CVodeFree(&cvode_mem);
206
207     return(0);
208 }
209
210
211 /*
212 *-----
213 * Functions called by the solver
214 *-----
215 */
216
217 /*
218 * f routine. Compute function f(t,y).
219 */
220
221 static int f(realtype t, N_Vector y, N_Vector ydot, void *f_data)
222 {
223     realtype y1, y2, y3, yd1, yd3;
224
225     y1 = Ith(y,1); y2 = Ith(y,2); y3 = Ith(y,3);
226
227     yd1 = Ith(ydot,1) = RCONST(-0.04)*y1 + RCONST(1.0e4)*y2*y3;
228     yd3 = Ith(ydot,3) = RCONST(3.0e7)*y2*y2;
229     Ith(ydot,2) = -yd1 - yd3;
230
231     return(0);
232 }
233
234 /*

```



```

294
295     return;
296 }
297
298 /*
299  * Get and print some final statistics
300  */
301
302 static void PrintFinalStats(void *cvode_mem)
303 {
304     long int nst, nfe, nsetups, nje, nfeLS, nni, ncnf, netf, nge;
305     int flag;
306
307     flag = CVodeGetNumSteps(cvode_mem, &nst);
308     check_flag(&flag, "CVodeGetNumSteps", 1);
309     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
310     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
311     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
312     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
313     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
314     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
315     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
316     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
317     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncnf);
318     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
319
320     flag = CVDenseGetNumJacEvals(cvode_mem, &nje);
321     check_flag(&flag, "CVDenseGetNumJacEvals", 1);
322     flag = CVDenseGetNumRhsEvals(cvode_mem, &nfeLS);
323     check_flag(&flag, "CVDenseGetNumRhsEvals", 1);
324
325     flag = CVodeGetNumGEvals(cvode_mem, &nge);
326     check_flag(&flag, "CVodeGetNumGEvals", 1);
327
328     printf("\nFinal Statistics:\n");
329     printf("nst=%-6ld_nfe=%-6ld_nsetups=%-6ld_nfeLS=%-6ld_nje=%-6ld\n",
330           nst, nfe, nsetups, nfeLS, nje);
331     printf("nni=%-6ld_ncnf=%-6ld_netf=%-6ld_nge=%-6ld\n\n",
332           nni, ncnf, netf, nge);
333 }
334
335 /*
336  * Check function return value...
337  *   opt == 0 means SUNDIALS function allocates memory so check if
338  *   returned NULL pointer
339  *   opt == 1 means SUNDIALS function returns a flag so check if
340  *   flag >= 0
341  *   opt == 2 means function allocates memory so check if returned
342  *   NULL pointer
343  */
344
345 static int check_flag(void *flagvalue, char *funcname, int opt)
346 {
347     int *errflag;
348
349     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
350     if (opt == 0 && flagvalue == NULL) {
351         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
352               funcname);

```

```

353     return(1); }
354
355     /* Check if flag < 0 */
356     else if (opt == 1) {
357         errflag = (int *) flagvalue;
358         if (*errflag < 0) {
359             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
360                     funcname, *errflag);
361             return(1); }}
362
363     /* Check if function returned NULL pointer - no memory allocated */
364     else if (opt == 2 && flagvalue == NULL) {
365         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
366                 funcname);
367         return(1); }
368
369     return(0);
370 }

```

B Listing of cvbanx.c

```

1  /*
2  * -----
3  * $Revision: 1.5 $
4  * $Date: 2006/03/23 01:21:39 $
5  * -----
6  * Programmer(s): Scott D. Cohen, Alan C. Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem:
10 *
11 * The following is a simple example problem with 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 cvbanx.c */
37
38 #include "ccode.h"          /* prototypes for CVODE fcts. and consts. */
39 #include "nvector_serial.h" /* serial N_Vector types, fcts., and macros */
40 #include "ccode_band.h"     /* prototype for CVBand */
41 #include "sundials_band.h"   /* definitions of type BandMat and macros */
42 #include "sundials_types.h"  /* definition of type realtype */
43 #include "sundials_math.h"   /* definition of ABS and EXP */
44
45 /* Problem Constants */
46
47 #define XMAX  RCONST(2.0) /* domain boundaries */
48 #define YMAX  RCONST(1.0)
49 #define MX    10          /* mesh dimensions */
50 #define MY    5
51 #define NEQ   MX*MY       /* number of equations */
52 #define ATOL  RCONST(1.0e-5) /* scalar absolute tolerance */
53 #define T0    RCONST(0.0) /* initial time */
54 #define T1    RCONST(0.1) /* first output time */
55 #define DTOUT RCONST(0.1) /* output time increment */
56 #define NOUT  10          /* number of output times */
57

```

```

58 #define ZERO RCONST(0.0)
59 #define HALF RCONST(0.5)
60 #define ONE RCONST(1.0)
61 #define TWO RCONST(2.0)
62 #define FIVE RCONST(5.0)
63
64 /* User-defined vector access macro IJth */
65
66 /* IJth is defined in order to isolate the translation from the
67    mathematical 2-dimensional structure of the dependent variable vector
68    to the underlying 1-dimensional storage.
69    IJth(vdata,i,j) references the element in the vdata array for
70    u at mesh point (i,j), where 1 <= i <= MX, 1 <= j <= MY.
71    The vdata array is obtained via the macro call vdata = NV_DATA_S(v),
72    where v is an N_Vector.
73    The variables are ordered by the y index j, then by the x index i. */
74
75 #define IJth(vdata,i,j) (vdata[(j-1) + (i-1)*MY])
76
77 /* Type : UserData (contains grid constants) */
78
79 typedef struct {
80     realtype dx, dy, hdcoef, hacoef, vdcoef;
81 } *UserData;
82
83 /* Private Helper Functions */
84
85 static void SetIC(N_Vector u, UserData data);
86 static void PrintHeader(realtype reltol, realtype abstol, realtype umax);
87 static void PrintOutput(realtype t, realtype umax, long int nst);
88 static void PrintFinalStats(void *cnode_mem);
89
90 /* Private function to check function return values */
91
92 static int check_flag(void *flagvalue, char *funcname, int opt);
93
94 /* Functions Called by the Solver */
95
96 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
97 static int Jac(long int N, long int mu, long int ml, BandMat J,
98               realtype t, N_Vector u, N_Vector fu, void *jac_data,
99               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
100
101 /*
102  *-----
103  * Main Program
104  *-----
105  */
106
107 int main(void)
108 {
109     realtype dx, dy, reltol, abstol, t, tout, umax;
110     N_Vector u;
111     UserData data;
112     void *cnode_mem;
113     int iout, flag;
114     long int nst;
115
116     u = NULL;

```

```

117     data = NULL;
118     ccode_mem = NULL;
119
120     /* Create a serial vector */
121
122     u = N_VNew_Serial(NEQ); /* Allocate u vector */
123     if(check_flag((void*)u, "N_VNew_Serial", 0)) return(1);
124
125     reltol = ZERO; /* Set the tolerances */
126     abstol = ATOL;
127
128     data = (UserData) malloc(sizeof *data); /* Allocate data memory */
129     if(check_flag((void *)data, "malloc", 2)) return(1);
130     dx = data->dx = XMAX/(MX+1); /* Set grid coefficients in data */
131     dy = data->dy = YMAX/(MY+1);
132     data->hdcoef = ONE/(dx*dx);
133     data->hacoef = HALF/(TWO*dx);
134     data->vdccoef = ONE/(dy*dy);
135
136     SetIC(u, data); /* Initialize u vector */
137
138     /*
139      Call CcodeCreate to create integrator memory
140
141      CV_BDF      specifies the Backward Differentiation Formula
142      CV_NEWTON   specifies a Newton iteration
143
144      A pointer to the integrator problem memory is returned and
145      stored in ccode_mem.
146     */
147
148     ccode_mem = CcodeCreate(CV_BDF, CV_NEWTON);
149     if(check_flag((void *)ccode_mem, "CcodeCreate", 0)) return(1);
150
151     /*
152      Call CcodeMalloc to initialize the integrator memory:
153
154      ccode_mem is the pointer to the integrator memory returned by CcodeCreate
155      f          is the user's right hand side function in y'=f(t,y)
156      T0         is the initial time
157      u          is the initial dependent variable vector
158      CV_SS      specifies scalar relative and absolute tolerances
159      reltol     is the scalar relative tolerance
160      &abstol   is a pointer to the scalar absolute tolerance
161     */
162
163     flag = CcodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
164     if(check_flag(&flag, "CcodeMalloc", 1)) return(1);
165
166     /* Set the pointer to user-defined data */
167
168     flag = CcodeSetFdata(ccode_mem, data);
169     if(check_flag(&flag, "CcodeSetFdata", 1)) return(1);
170
171     /* Call CVBand to specify the CVBAND band linear solver */
172
173     flag = CVBand(ccode_mem, NEQ, MY, MY);
174     if(check_flag(&flag, "CVBand", 1)) return(1);
175

```



```

176  /* Set the user-supplied Jacobian routine Jac and
177     the pointer to the user-defined block data. */
178
179  flag = CVBandSetJacFn(cvode_mem, Jac, data);
180  if(check_flag(&flag, "CVBandSetJacFn", 1)) return(1);
181
182  /* In loop over output points: call CVode, print results, test for errors */
183
184  umax = N_VMaxNorm(u);
185  PrintHeader(reltol, abstol, umax);
186  for(iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {
187      flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
188      if(check_flag(&flag, "CVode", 1)) break;
189      umax = N_VMaxNorm(u);
190      flag = CVodeGetNumSteps(cvode_mem, &nst);
191      check_flag(&flag, "CVodeGetNumSteps", 1);
192      PrintOutput(t, umax, nst);
193  }
194
195  PrintFinalStats(cvode_mem); /* Print some final statistics */
196
197  N_VDestroy_Serial(u); /* Free the u vector */
198  CVodeFree(&cvode_mem); /* Free the integrator memory */
199  free(data); /* Free the user data */
200
201  return(0);
202 }
203
204 /*
205  *-----
206  * Functions called by the solver
207  *-----
208  */
209
210 /* f routine. Compute f(t,u). */
211
212 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
213 {
214     realtype uij, udn, uup, ult, urt, hordc, horac, verdc, hdiff, hadv, vdiff;
215     realtype *udata, *dudata;
216     int i, j;
217     UserData data;
218
219     udata = NV_DATA_S(u);
220     dudata = NV_DATA_S(udot);
221
222     /* Extract needed constants from data */
223
224     data = (UserData) f_data;
225     hordc = data->hdcoef;
226     horac = data->hacoe;
227     verdc = data->vdcoef;
228
229     /* Loop over all grid points. */
230
231     for (j=1; j <= MY; j++) {
232
233         for (i=1; i <= MX; i++) {

```

```

235     /* Extract u at x_i, y_j and four neighboring points */
236
237     uij = IJth(udata, i, j);
238     udn = (j == 1) ? ZERO : IJth(udata, i, j-1);
239     uup = (j == MY) ? ZERO : IJth(udata, i, j+1);
240     ult = (i == 1) ? ZERO : IJth(udata, i-1, j);
241     urt = (i == MX) ? ZERO : IJth(udata, i+1, j);
242
243     /* Set diffusion and advection terms and load into udot */
244
245     hdiff = hordc*(ult - TWO*uij + urt);
246     hadv = horac*(urt - ult);
247     vdiff = verdc*(uup - TWO*uij + udn);
248     IJth(dudata, i, j) = hdiff + hadv + vdiff;
249 }
250 }
251
252     return(0);
253 }
254
255 /* Jacobian routine. Compute J(t,u). */
256
257 static int Jac(long int N, long int mu, long int ml, BandMat J,
258               realtype t, N_Vector u, N_Vector fu, void *jac_data,
259               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
260 {
261     long int i, j, k;
262     realtype *kthCol, hordc, horac, verdc;
263     UserData data;
264
265     /*
266      The components of f = udot that depend on u(i,j) are
267      f(i,j), f(i-1,j), f(i+1,j), f(i,j-1), f(i,j+1), with
268      df(i,j)/du(i,j) = -2 (1/dx^2 + 1/dy^2)
269      df(i-1,j)/du(i,j) = 1/dx^2 + .25/dx (if i > 1)
270      df(i+1,j)/du(i,j) = 1/dx^2 - .25/dx (if i < MX)
271      df(i,j-1)/du(i,j) = 1/dy^2 (if j > 1)
272      df(i,j+1)/du(i,j) = 1/dy^2 (if j < MY)
273     */
274
275     data = (UserData) jac_data;
276     hordc = data->hdcoef;
277     horac = data->haccoef;
278     verdc = data->vdcoef;
279
280     for (j=1; j <= MY; j++) {
281         for (i=1; i <= MX; i++) {
282             k = j-1 + (i-1)*MY;
283             kthCol = BAND_COL(J,k);
284
285             /* set the kth column of J */
286
287             BAND_COL_ELEM(kthCol,k,k) = -TWO*(verdc+hordc);
288             if (i != 1) BAND_COL_ELEM(kthCol,k-MY,k) = hordc + horac;
289             if (i != MX) BAND_COL_ELEM(kthCol,k+MY,k) = hordc - horac;
290             if (j != 1) BAND_COL_ELEM(kthCol,k-1,k) = verdc;
291             if (j != MY) BAND_COL_ELEM(kthCol,k+1,k) = verdc;
292         }
293     }

```

```

294
295     return(0);
296 }
297
298 /*
299  *-----
300  * Private helper functions
301  *-----
302  */
303
304 /* Set initial conditions in u vector */
305
306 static void SetIC(N_Vector u, UserData data)
307 {
308     int i, j;
309     realtype x, y, dx, dy;
310     realtype *udata;
311
312     /* Extract needed constants from data */
313
314     dx = data->dx;
315     dy = data->dy;
316
317     /* Set pointer to data array in vector u. */
318
319     udata = NV_DATA_S(u);
320
321     /* Load initial profile into u vector */
322
323     for (j=1; j <= MY; j++) {
324         y = j*dy;
325         for (i=1; i <= MX; i++) {
326             x = i*dx;
327             IJth(udata,i,j) = x*(XMAX - x)*y*(YMAX - y)*EXP(FIVE*x*y);
328         }
329     }
330 }
331
332 /* Print first lines of output (problem description) */
333
334 static void PrintHeader(realtype reltol, realtype abstol, realtype umax)
335 {
336     printf("\n2-D Advection-Diffusion Equation\n");
337     printf("Mesh dimensions = %d X %d\n", MX, MY);
338     printf("Total system size = %d\n", NEQ);
339     #if defined(SUNDIALS_EXTENDED_PRECISION)
340     printf("Tolerance parameters: reltol = %Lg, abstol = %Lg\n\n", reltol, abstol);
341     printf("At t = %Lg, max.norm(u) = %14.6Le\n", T0, umax);
342     #elif defined(SUNDIALS_DOUBLE_PRECISION)
343     printf("Tolerance parameters: reltol = %lg, abstol = %lg\n\n", reltol, abstol);
344     printf("At t = %lg, max.norm(u) = %14.6le\n", T0, umax);
345     #else
346     printf("Tolerance parameters: reltol = %g, abstol = %g\n\n", reltol, abstol);
347     printf("At t = %g, max.norm(u) = %14.6e\n", T0, umax);
348     #endif
349
350     return;
351 }
352

```

```

353  /* Print current value */
354
355  static void PrintOutput(realtype t, realtype umax, long int nst)
356  {
357      #if defined(SUNDIALS_EXTENDED_PRECISION)
358          printf("At_t=%4.2Lf_uu_max.norm(u)_=%14.6Le_uu_nst=%4ld\n", t, umax, nst);
359      #elif defined(SUNDIALS_DOUBLE_PRECISION)
360          printf("At_t=%4.2f_uu_max.norm(u)_=%14.6le_uu_nst=%4ld\n", t, umax, nst);
361      #else
362          printf("At_t=%4.2f_uu_max.norm(u)_=%14.6e_uu_nst=%4ld\n", t, umax, nst);
363      #endif
364
365      return;
366  }
367
368  /* Get and print some final statistics */
369
370  static void PrintFinalStats(void *cvode_mem)
371  {
372      int flag;
373      long int nst, nfe, nsetups, netf, nni, ncnf, nje, nfeLS;
374
375      flag = CVodeGetNumSteps(cvode_mem, &nst);
376      check_flag(&flag, "CVodeGetNumSteps", 1);
377      flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
378      check_flag(&flag, "CVodeGetNumRhsEvals", 1);
379      flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
380      check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
381      flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
382      check_flag(&flag, "CVodeGetNumErrTestFails", 1);
383      flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
384      check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
385      flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncnf);
386      check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
387
388      flag = CVBandGetNumJacEvals(cvode_mem, &nje);
389      check_flag(&flag, "CVBandGetNumJacEvals", 1);
390      flag = CVBandGetNumRhsEvals(cvode_mem, &nfeLS);
391      check_flag(&flag, "CVBandGetNumRhsEvals", 1);
392
393      printf("\nFinal_Statistics:\n");
394      printf("nst=%-6ld_nfe=%-6ld_nsetups=%-6ld_nfeLS=%-6ld_nje=%ld\n",
395            nst, nfe, nsetups, nfeLS, nje);
396      printf("nni=%-6ld_ncnf=%-6ld_netf=%ld\n_n\n",
397            nni, ncnf, netf);
398
399      return;
400  }
401
402  /* Check function return value...
403     opt == 0 means SUNDIALS function allocates memory so check if
404         returned NULL pointer
405     opt == 1 means SUNDIALS function returns a flag so check if
406         flag >= 0
407     opt == 2 means function allocates memory so check if returned
408         NULL pointer */
409
410  static int check_flag(void *flagvalue, char *funcname, int opt)
411  {

```

```

412     int *errflag;
413
414     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
415
416     if (opt == 0 && flagvalue == NULL) {
417         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
418             funcname);
419         return(1); }
420
421     /* Check if flag < 0 */
422
423     else if (opt == 1) {
424         errflag = (int *) flagvalue;
425         if (*errflag < 0) {
426             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
427                 funcname, *errflag);
428             return(1); }}
429
430     /* Check if function returned NULL pointer - no memory allocated */
431
432     else if (opt == 2 && flagvalue == NULL) {
433         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
434             funcname);
435         return(1); }
436
437     return(0);
438 }

```

C Listing of cvkryx.c

```

1  /*
2  * -----
3  * $Revision: 1.6 $
4  * $Date: 2006/03/23 01:21:39 $
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
39 #include "cvmde.h" /* main integrator header file */
40 #include "nvector_serial.h" /* serial N_Vector types, fct. and macros */
41 #include "cvmde_spgmr.h" /* prototypes & constants for CVSPGMR solver */
42 #include "sundials_smalldense.h" /* use generic DENSE solver in preconditioning */
43 #include "sundials_types.h" /* definition of realtype */
44 #include "sundials_math.h" /* contains the macros ABS, SQR, and EXP */
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 /* CNodeMalloc 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 int 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     cnode_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     is the relative tolerance
195      &abstol    is a pointer to the scalar absolute tolerance      */
196  flag = CcodeMalloc(ccode_mem, f, T0, u, CV_SS, reltol, &abstol);
197  if(check_flag(&flag, "CcodeMalloc", 1)) return(1);
198
199  /* Call CVSpgmr to specify the linear solver CVSPGMR
200      with left preconditioning and the maximum Krylov dimension maxl */
201  flag = CVSpgmr(ccode_mem, PREC_LEFT, 0);
202  if(check_flag(&flag, "CVSpgmr", 1)) return(1);
203
204  /* Set modified Gram-Schmidt orthogonalization, preconditioner
205      setup and solve routines Precond and PSolve, and the pointer
206      to the user-defined block data */
207  flag = CVSpilsSetGSType(ccode_mem, MODIFIED_GS);
208  if(check_flag(&flag, "CVSpilsSetGSType", 1)) return(1);
209
210  flag = CVSpilsSetPreconditioner(ccode_mem, Precond, PSolve, data);
211  if(check_flag(&flag, "CVSpilsSetPreconditioner", 1)) return(1);
212
213  /* In loop over output points, call Ccode, print results, test for error */
214  printf("\n2-species diurnal advection-diffusion problem\n\n");
215  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
216      flag = Ccode(ccode_mem, tout, u, &t, CV_NORMAL);
217      PrintOutput(ccode_mem, u, t);
218      if(check_flag(&flag, "Ccode", 1)) break;
219  }
220
221  PrintFinalStats(ccode_mem);
222
223  /* Free memory */
224  N_VDestroy_Serial(u);
225  FreeUserData(data);
226  CcodeFree(&ccode_mem);
227
228  return(0);
229 }
230
231 /*
232  *-----
233  * Private helper functions

```

```

234  *-----
235  */
236
237  /* Allocate memory for data structure of type UserData */
238
239  static UserData AllocUserData(void)
240  {
241      int jx, jy;
242      UserData data;
243
244      data = (UserData) malloc(sizeof *data);
245
246      for (jx=0; jx < MX; jx++) {
247          for (jy=0; jy < MY; jy++) {
248              (data->P)[jx][jy] = denalloc(NUM_SPECIES);
249              (data->Jbd)[jx][jy] = denalloc(NUM_SPECIES);
250              (data->pivot)[jx][jy] = denallocpiv(NUM_SPECIES);
251          }
252      }
253
254      return(data);
255  }
256
257  /* Load problem constants in data */
258
259  static void InitUserData(UserData data)
260  {
261      data->om = PI/HALFDAY;
262      data->dx = (XMAX-XMIN)/(MX-1);
263      data->dy = (YMAX-YMIN)/(MY-1);
264      data->hdco = KH/SQR(data->dx);
265      data->haco = VEL/(TWO*data->dx);
266      data->vdco = (ONE/SQR(data->dy))*KV0;
267  }
268
269  /* Free data memory */
270
271  static void FreeUserData(UserData data)
272  {
273      int jx, jy;
274
275      for (jx=0; jx < MX; jx++) {
276          for (jy=0; jy < MY; jy++) {
277              denfree((data->P)[jx][jy]);
278              denfree((data->Jbd)[jx][jy]);
279              denfreepiv((data->pivot)[jx][jy]);
280          }
281      }
282
283      free(data);
284  }
285
286  /* Set initial conditions in u */
287
288  static void SetInitialProfiles(N_Vector u, realtype dx, realtype dy)
289  {
290      int jx, jy;
291      realtype x, y, cx, cy;
292      realtype *udata;

```

```

293
294 /* Set pointer to data array in vector u. */
295
296 udata = NV_DATA_S(u);
297
298 /* Load initial profiles of c1 and c2 into u vector */
299
300 for (jy=0; jy < MY; jy++) {
301     y = YMIN + jy*dy;
302     cy = SQR(RCONST(0.1)*(y - YMID));
303     cy = ONE - cy + RCONST(0.5)*SQR(cy);
304     for (jx=0; jx < MX; jx++) {
305         x = XMIN + jx*dx;
306         cx = SQR(RCONST(0.1)*(x - XMID));
307         cx = ONE - cx + RCONST(0.5)*SQR(cx);
308         IJKth(udata,1,jx,jy) = C1_SCALE*cx*cy;
309         IJKth(udata,2,jx,jy) = C2_SCALE*cx*cy;
310     }
311 }
312 }
313
314 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
315
316 static void PrintOutput(void *cnode_mem, N_Vector u, realtype t)
317 {
318     long int nst;
319     int qu, flag;
320     realtype hu, *udata;
321     int mxh = MX/2 - 1, myh = MY/2 - 1, mx1 = MX - 1, my1 = MY - 1;
322
323     udata = NV_DATA_S(u);
324
325     flag = CVodeGetNumSteps(cnode_mem, &nst);
326     check_flag(&flag, "CVodeGetNumSteps", 1);
327     flag = CVodeGetLastOrder(cnode_mem, &qu);
328     check_flag(&flag, "CVodeGetLastOrder", 1);
329     flag = CVodeGetLastStep(cnode_mem, &hu);
330     check_flag(&flag, "CVodeGetLastStep", 1);
331
332     #if defined(SUNDIALS_EXTENDED_PRECISION)
333     printf("t=%%.2Le%%no. steps=%ld%%order=%d%%stepsize=%%.2Le\n",
334         t, nst, qu, hu);
335     printf("c1_(bot.left/middle/top_rtr.)=%%.12.3Le%%%.12.3Le%%%.12.3Le\n",
336         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
337     printf("c2_(bot.left/middle/top_rtr.)=%%.12.3Le%%%.12.3Le%%%.12.3Le\n\n",
338         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
339     #elif defined(SUNDIALS_DOUBLE_PRECISION)
340     printf("t=%%.2le%%no. steps=%ld%%order=%d%%stepsize=%%.2le\n",
341         t, nst, qu, hu);
342     printf("c1_(bot.left/middle/top_rtr.)=%%.12.3le%%%.12.3le%%%.12.3le\n",
343         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
344     printf("c2_(bot.left/middle/top_rtr.)=%%.12.3le%%%.12.3le%%%.12.3le\n\n",
345         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
346     #else
347     printf("t=%%.2e%%no. steps=%ld%%order=%d%%stepsize=%%.2e\n",
348         t, nst, qu, hu);
349     printf("c1_(bot.left/middle/top_rtr.)=%%.12.3e%%%.12.3e%%%.12.3e\n",
350         IJKth(udata,1,0,0), IJKth(udata,1,mxh,myh), IJKth(udata,1,mx1,my1));
351     printf("c2_(bot.left/middle/top_rtr.)=%%.12.3e%%%.12.3e%%%.12.3e\n\n",

```

```

352         IJKth(udata,2,0,0), IJKth(udata,2,mxh,myh), IJKth(udata,2,mx1,my1));
353 #endif
354 }
355
356 /* Get and print final statistics */
357
358 static void PrintFinalStats(void *cvode_mem)
359 {
360     long int lenrw, leniw ;
361     long int lenrwLS, leniwLS;
362     long int nst, nfe, nsetups, nni, ncnf, netf;
363     long int nli, npe, nps, ncfl, nfeLS;
364     int flag;
365
366     flag = CVodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
367     check_flag(&flag, "CVodeGetWorkSpace", 1);
368     flag = CVodeGetNumSteps(cvode_mem, &nst);
369     check_flag(&flag, "CVodeGetNumSteps", 1);
370     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
371     check_flag(&flag, "CVodeGetNumRhsEvals", 1);
372     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
373     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1);
374     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
375     check_flag(&flag, "CVodeGetNumErrTestFails", 1);
376     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
377     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1);
378     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncnf);
379     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1);
380
381     flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
382     check_flag(&flag, "CVSpilsGetWorkSpace", 1);
383     flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
384     check_flag(&flag, "CVSpilsGetNumLinIters", 1);
385     flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
386     check_flag(&flag, "CVSpilsGetNumPrecEvals", 1);
387     flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
388     check_flag(&flag, "CVSpilsGetNumPrecSolves", 1);
389     flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
390     check_flag(&flag, "CVSpilsGetNumConvFails", 1);
391     flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
392     check_flag(&flag, "CVSpilsGetNumRhsEvals", 1);
393
394     printf("\nFinal Statistics...\n\n");
395     printf("lenrw=%5ld leniw=%5ld\n", lenrw, leniw);
396     printf("lenrwLS=%5ld leniwLS=%5ld\n", lenrwLS, leniwLS);
397     printf("nst=%5ld", nst);
398     printf("nfe=%5ld nfeLS=%5ld", nfe, nfeLS);
399     printf("nni=%5ld nli=%5ld", nni, nli);
400     printf("nsetups=%5ld netf=%5ld", nsetups, netf);
401     printf("npe=%5ld nps=%5ld", npe, nps);
402     printf("ncnf=%5ld ncfl=%5ld", ncnf, ncfl);
403 }
404
405 /* Check function return value...
406     opt == 0 means SUNDIALS function allocates memory so check if
407         returned NULL pointer
408     opt == 1 means SUNDIALS function returns a flag so check if
409         flag >= 0
410     opt == 2 means function allocates memory so check if returned

```

```

411         NULL pointer */
412
413 static int check_flag(void *flagvalue, char *funcname, int opt)
414 {
415     int *errflag;
416
417     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
418     if (opt == 0 && flagvalue == NULL) {
419         fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
420             funcname);
421         return(1); }
422
423     /* Check if flag < 0 */
424     else if (opt == 1) {
425         errflag = (int *) flagvalue;
426         if (*errflag < 0) {
427             fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
428                 funcname, *errflag);
429             return(1); }}
430
431     /* Check if function returned NULL pointer - no memory allocated */
432     else if (opt == 2 && flagvalue == NULL) {
433         fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
434             funcname);
435         return(1); }
436
437     return(0);
438 }
439
440 /*
441  *-----
442  * Functions called by the solver
443  *-----
444  */
445
446 /* f routine. Compute RHS function f(t,u). */
447
448 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
449 {
450     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
451     realtype clrt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
452     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
453     realtype q4coef, dely, verdco, hordco, horaco;
454     realtype *udata, *dudata;
455     int jx, jy, idn, iup, ileft, irect;
456     UserData data;
457
458     data = (UserData) f_data;
459     udata = NV_DATA_S(u);
460     dudata = NV_DATA_S(udot);
461
462     /* Set diurnal rate coefficients. */
463
464     s = sin(data->om*t);
465     if (s > ZERO) {
466         q3 = EXP(-A3/s);
467         data->q4 = EXP(-A4/s);
468     } else {
469         q3 = ZERO;

```

```

470     data->q4 = ZERO;
471 }
472
473 /* Make local copies of problem variables, for efficiency. */
474
475 q4coef = data->q4;
476 dely = data->dy;
477 verdco = data->vdco;
478 hordco = data->hdco;
479 horaco = data->haco;
480
481 /* Loop over all grid points. */
482
483 for (jy=0; jy < MY; jy++) {
484
485     /* Set vertical diffusion coefficients at jy +/- 1/2 */
486
487     ydn = YMIN + (jy - RCONST(0.5))*dely;
488     yup = ydn + dely;
489     cydn = verdco*EXP(RCONST(0.2)*ydn);
490     cyup = verdco*EXP(RCONST(0.2)*yup);
491     idn = (jy == 0) ? 1 : -1;
492     iup = (jy == MY-1) ? -1 : 1;
493     for (jx=0; jx < MX; jx++) {
494
495         /* Extract c1 and c2, and set kinetic rate terms. */
496
497         c1 = IJKth(udata,1,jx,jy);
498         c2 = IJKth(udata,2,jx,jy);
499         qq1 = Q1*c1*C3;
500         qq2 = Q2*c1*c2;
501         qq3 = q3*C3;
502         qq4 = q4coef*c2;
503         rkin1 = -qq1 - qq2 + TWO*qq3 + qq4;
504         rkin2 = qq1 - qq2 - qq4;
505
506         /* Set vertical diffusion terms. */
507
508         c1dn = IJKth(udata,1,jx,jy+idn);
509         c2dn = IJKth(udata,2,jx,jy+idn);
510         c1up = IJKth(udata,1,jx,jy+iup);
511         c2up = IJKth(udata,2,jx,jy+iup);
512         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
513         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
514
515         /* Set horizontal diffusion and advection terms. */
516
517         ileft = (jx == 0) ? 1 : -1;
518         irect = (jx == MX-1) ? -1 : 1;
519         c1lt = IJKth(udata,1,jx+ileft,jy);
520         c2lt = IJKth(udata,2,jx+ileft,jy);
521         c1rt = IJKth(udata,1,jx+irect,jy);
522         c2rt = IJKth(udata,2,jx+irect,jy);
523         hord1 = hordco*(c1rt - TWO*c1 + c1lt);
524         hord2 = hordco*(c2rt - TWO*c2 + c2lt);
525         horad1 = horaco*(c1rt - c1lt);
526         horad2 = horaco*(c2rt - c2lt);
527
528         /* Load all terms into udot. */

```

```

529         IJKth(dudata, 1, jx, jy) = vertd1 + hord1 + horad1 + rkin1;
530         IJKth(dudata, 2, jx, jy) = vertd2 + hord2 + horad2 + rkin2;
531     }
532 }
533 }
534
535     return(0);
536 }
537
538 /* Preconditioner setup routine. Generate and preprocess P. */
539
540 static int Precond(realtype tn, N_Vector u, N_Vector fu,
541                   booleantype jok, booleantype *jcurPtr, realtype gamma,
542                   void *P_data, N_Vector vtemp1, N_Vector vtemp2,
543                   N_Vector vtemp3)
544 {
545     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
546     realtype **(*P)[MY], **(*Jbd)[MY];
547     long int **(*pivot)[MY], ier;
548     int jx, jy;
549     realtype *udata, **a, **j;
550     UserData data;
551
552     /* Make local copies of pointers in P_data, and of pointer to u's data */
553
554     data = (UserData) P_data;
555     P = data->P;
556     Jbd = data->Jbd;
557     pivot = data->pivot;
558     udata = NV_DATA_S(u);
559
560     if (jok) {
561
562         /* jok = TRUE: Copy Jbd to P */
563
564         for (jy=0; jy < MY; jy++)
565             for (jx=0; jx < MX; jx++)
566                 dencopy(Jbd[jx][jy], P[jx][jy], NUM_SPECIES);
567
568         *jcurPtr = FALSE;
569
570     }
571
572     else {
573         /* jok = FALSE: Generate Jbd from scratch and copy to P */
574
575         /* Make local copies of problem variables, for efficiency. */
576
577         q4coef = data->q4;
578         dely = data->dy;
579         verdco = data->vdco;
580         hordco = data->hdco;
581
582         /* Compute 2x2 diagonal Jacobian blocks (using q4 values
583            computed on the last f call). Load into P. */
584
585         for (jy=0; jy < MY; jy++) {
586             ydn = YMIN + (jy - RCONST(0.5))*dely;
587             yup = ydn + dely;

```

```

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

```



```

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

```

D Listing of cvnonx_p.c

```

1  /*
2  * -----
3  * $Revision: 1.7 $
4  * $Date: 2006/03/23 21:44:45 $
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
37 #include "cvsode.h"          /* prototypes for CVODE fcts. */
38 #include "nvector_parallel.h" /* definition of N_Vector and macros */
39 #include "sundials_types.h"  /* definition of realtype */
40 #include "sundials_math.h"   /* definition of EXP */
41
42 #include "mpi.h"             /* MPI constants and types */
43
44 /* Problem Constants */
45
46 #define ZERO    RCONST(0.0)
47
48 #define XMAX    RCONST(2.0) /* domain boundary */
49 #define MX      10          /* mesh dimension */
50 #define NEQ     MX          /* number of equations */
51 #define ATOL    RCONST(1.0e-5) /* scalar absolute tolerance */
52 #define T0      ZERO        /* initial time */
53 #define T1      RCONST(0.5) /* first output time */
54 #define DTOUT   RCONST(0.5) /* output time increment */
55 #define NOUT    10          /* number of output times */
56
57 /* Type : UserData

```

```

58     contains grid constants, parallel machine parameters, work array. */
59
60 typedef struct {
61     realtype dx, hdcoef, hacoef;
62     int npes, my_pe;
63     MPI_Comm comm;
64     realtype z[100];
65 } *UserData;
66
67 /* Private Helper Functions */
68
69 static void SetIC(N_Vector u, realtype dx, long int my_length,
70                 long int my_base);
71
72 static void PrintIntro(int npes);
73
74 static void PrintData(realtype t, realtype umax, long int nst);
75
76 static void PrintFinalStats(void *cnode_mem);
77
78 /* Functions Called by the Solver */
79
80 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
81
82 /* Private function to check function return values */
83
84 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
85
86 /***** Main Program *****/
87
88 int main(int argc, char *argv[])
89 {
90     realtype dx, reltol, abstol, t, tout, umax;
91     N_Vector u;
92     UserData data;
93     void *cnode_mem;
94     int iout, flag, my_pe, npes;
95     long int local_N, nperpe, nrem, my_base, nst;
96
97     MPI_Comm comm;
98
99     u = NULL;
100    data = NULL;
101    cnode_mem = NULL;
102
103    /* Get processor number, total number of pe's, and my_pe. */
104    MPI_Init(&argc, &argv);
105    comm = MPI_COMM_WORLD;
106    MPI_Comm_size(comm, &npes);
107    MPI_Comm_rank(comm, &my_pe);
108
109    /* Set local vector length. */
110    nperpe = NEQ/npes;
111    nrem = NEQ - npes*nperpe;
112    local_N = (my_pe < nrem) ? nperpe+1 : nperpe;
113    my_base = (my_pe < nrem) ? my_pe*local_N : my_pe*nperpe + nrem;
114
115    data = (UserData) malloc(sizeof *data); /* Allocate data memory */
116    if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);

```

```

117
118 data->comm = comm;
119 data->npes = npes;
120 data->my_pe = my_pe;
121
122 u = N_VNew_Parallel(comm, local_N, NEQ); /* Allocate u vector */
123 if(check_flag((void *)u, "N_VNew", 0, my_pe)) MPI_Abort(comm, 1);
124
125 reltol = ZERO; /* Set the tolerances */
126 abstol = ATOL;
127
128 dx = data->dx = XMAX/((realtype)(MX+1)); /* Set grid coefficients in data */
129 data->hdcoef = RCONST(1.0)/(dx*dx);
130 data->hacoef = RCONST(0.5)/(RCONST(2.0)*dx);
131
132 SetIC(u, dx, local_N, my_base); /* Initialize u vector */
133
134 /*
135  Call CNodeCreate to create the solver memory:
136
137  CV_ADAMS specifies the Adams Method
138  CV_FUNCTIONAL specifies functional iteration
139
140  A pointer to the integrator memory is returned and stored in cnode_mem.
141 */
142
143 cnode_mem = CNodeCreate(CV_ADAMS, CV_FUNCTIONAL);
144 if(check_flag((void *)cnode_mem, "CNodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
145
146 flag = CNodeSetFdata(cnode_mem, data);
147 if(check_flag(&flag, "CNodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
148
149 /*
150  Call CNodeMalloc to initialize the integrator memory:
151
152  cnode_mem is the pointer to the integrator memory returned by CNodeCreate
153  f is the user's right hand side function in y'=f(t,y)
154  T0 is the initial time
155  u is the initial dependent variable vector
156  CV_SS specifies scalar relative and absolute tolerances
157  reltol is the relative tolerance
158  &abstol is a pointer to the scalar absolute tolerance
159 */
160
161 flag = CNodeMalloc(cnode_mem, f, T0, u, CV_SS, reltol, &abstol);
162 if(check_flag(&flag, "CNodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
163
164 if (my_pe == 0) PrintIntro(npes);
165
166 umax = N_VMaxNorm(u);
167
168 if (my_pe == 0) {
169     t = T0;
170     PrintData(t, umax, 0);
171 }
172
173 /* In loop over output points, call CNode, print results, test for error */
174
175 for (iout=1, tout=T1; iout <= NOUT; iout++, tout += DTOUT) {

```

```

176     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
177     if(check_flag(&flag, "CVode", 1, my_pe)) break;
178     umax = N_VMaxNorm(u);
179     flag = CVodeGetNumSteps(cvode_mem, &nst);
180     check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
181     if (my_pe == 0) PrintData(t, umax, nst);
182 }
183
184 if (my_pe == 0)
185     PrintFinalStats(cvode_mem); /* Print some final statistics */
186
187 N_VDestroy_Parallel(u); /* Free the u vector */
188 CVodeFree(&cvode_mem); /* Free the integrator memory */
189 free(data); /* Free user data */
190
191 MPI_Finalize();
192
193 return(0);
194 }
195
196 /***** Private Helper Functions *****/
197
198 /* Set initial conditions in u vector */
199
200 static void SetIC(N_Vector u, realtype dx, long int my_length,
201                  long int my_base)
202 {
203     int i;
204     long int iglobal;
205     realtype x;
206     realtype *udata;
207
208     /* Set pointer to data array and get local length of u. */
209     udata = NV_DATA_P(u);
210     my_length = NV_LOCLENGTH_P(u);
211
212     /* Load initial profile into u vector */
213     for (i=1; i<=my_length; i++) {
214         iglobal = my_base + i;
215         x = iglobal*dx;
216         udata[i-1] = x*(XMAX - x)*EXP(RCONST(2.0)*x);
217     }
218 }
219
220 /* Print problem introduction */
221
222 static void PrintIntro(int npes)
223 {
224     printf("\n1-D advection-diffusion equation, mesh size = %3d\n", MX);
225     printf("\nNumber of PEs = %3d\n", npes);
226
227     return;
228 }
229
230 /* Print data */
231
232 static void PrintData(realtype t, realtype umax, long int nst)
233 {
234

```

```

235 #if defined(SUNDIALS_EXTENDED_PRECISION)
236     printf("At_t=%4.2Lf_u_max.norm(u)=%14.6Le_u_nst=%4ld\n", t, umax, nst);
237 #elif defined(SUNDIALS_DOUBLE_PRECISION)
238     printf("At_t=%4.2f_u_max.norm(u)=%14.6le_u_nst=%4ld\n", t, umax, nst);
239 #else
240     printf("At_t=%4.2f_u_max.norm(u)=%14.6e_u_nst=%4ld\n", t, umax, nst);
241 #endif
242
243     return;
244 }
245
246 /* Print some final statistics located in the iopt array */
247
248 static void PrintFinalStats(void *cvode_mem)
249 {
250     long int nst, nfe, nni, ncnf, netf;
251     int flag;
252
253     flag = CVodeGetNumSteps(cvode_mem, &nst);
254     check_flag(&flag, "CVodeGetNumSteps", 1, 0);
255     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
256     check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
257     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
258     check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
259     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
260     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
261     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncnf);
262     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
263
264     printf("\nFinal_Statistics:\n\n");
265     printf("nst=%-6ld_u_nfe=%-6ld_u", nst, nfe);
266     printf("nni=%-6ld_u_ncnf=%-6ld_u_netf=%ld\n", nni, ncnf, netf);
267 }
268
269 /****** Function Called by the Solver *****/
270
271 /* f routine. Compute f(t,u). */
272
273 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
274 {
275     realtype ui, ult, urt, hordc, horac, hdiff, hadv;
276     realtype *udata, *dudata, *z;
277     int i;
278     int npes, my_pe, my_length, my_pe_m1, my_pe_p1, last_pe, my_last;
279     UserData data;
280     MPI_Status status;
281     MPI_Comm comm;
282
283     udata = NV_DATA_P(u);
284     dudata = NV_DATA_P(udot);
285
286     /* Extract needed problem constants from data */
287     data = (UserData) f_data;
288     hordc = data->hdcoef;
289     horac = data->haccoef;
290
291     /* Extract parameters for parallel computation. */
292     comm = data->comm;
293     npes = data->npes;          /* Number of processes. */

```

```

294 my_pe = data->my_pe;          /* Current process number. */
295 my_length = NV_LOCLENGTH_P(u); /* Number of local elements of u. */
296 z = data->z;
297
298 /* Compute related parameters. */
299 my_pe_m1 = my_pe - 1;
300 my_pe_p1 = my_pe + 1;
301 last_pe = npes - 1;
302 my_last = my_length - 1;
303
304 /* Store local segment of u in the working array z. */
305 for (i = 1; i <= my_length; i++)
306     z[i] = udata[i - 1];
307
308 /* Pass needed data to processes before and after current process. */
309 if (my_pe != 0)
310     MPI_Send(&z[1], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm);
311 if (my_pe != last_pe)
312     MPI_Send(&z[my_length], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm);
313
314 /* Receive needed data from processes before and after current process. */
315 if (my_pe != 0)
316     MPI_Recv(&z[0], 1, PVEC_REAL_MPI_TYPE, my_pe_m1, 0, comm, &status);
317 else z[0] = ZERO;
318 if (my_pe != last_pe)
319     MPI_Recv(&z[my_length+1], 1, PVEC_REAL_MPI_TYPE, my_pe_p1, 0, comm,
320             &status);
321 else z[my_length + 1] = ZERO;
322
323 /* Loop over all grid points in current process. */
324 for (i=1; i<=my_length; i++) {
325
326     /* Extract u at x_i and two neighboring points */
327     ui = z[i];
328     ult = z[i-1];
329     urt = z[i+1];
330
331     /* Set diffusion and advection terms and load into udot */
332     hdiff = hordc*(ult - RCONST(2.0)*ui + urt);
333     hadv = horac*(urt - ult);
334     dudata[i-1] = hdiff + hadv;
335 }
336
337 return(0);
338 }
339
340 /* Check function return value...
341     opt == 0 means SUNDIALS function allocates memory so check if
342         returned NULL pointer
343     opt == 1 means SUNDIALS function returns a flag so check if
344         flag >= 0
345     opt == 2 means function allocates memory so check if returned
346         NULL pointer */
347
348 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
349 {
350     int *errflag;
351
352     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */

```

```

353     if (opt == 0 && flagvalue == NULL) {
354         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
355             id, funcname);
356         return(1); }
357
358     /* Check if flag < 0 */
359     else if (opt == 1) {
360         errflag = (int *) flagvalue;
361         if (*errflag < 0) {
362             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
363                 id, funcname, *errflag);
364             return(1); }}
365
366     /* Check if function returned NULL pointer - no memory allocated */
367     else if (opt == 2 && flagvalue == NULL) {
368         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
369             id, funcname);
370         return(1); }
371
372     return(0);
373 }

```


E Listing of cvkryx_p.c

```

1  /*
2  * -----
3  * $Revision: 1.6 $
4  * $Date: 2006/03/23 01:21:38 $
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 cvkryx_p with N = NPEX*NPEY (see
47 * constants below).
48 * -----
49 */
50
51 #include <stdio.h>
52 #include <stdlib.h>
53 #include <math.h>
54
55 #include "cvmde.h" /* prototypes for CVODE fcts. */
56 #include "nvector_parallel.h" /* definition N_Vector and macro NV_DATA_P */
57 #include "cvmde_spgmr.h" /* prototypes and constants for CVSPGMR solver */

```

```

58 #include "sundials_smallldense.h" /* prototypes for small dense matrix fcts. */
59 #include "sundials_types.h"      /* definitions of realtype, booleantype */
60 #include "sundials_math.h"       /* definition of macros SQR and EXP */
61
62 #include "mpi.h"                  /* MPI constants and types */
63
64 /* Problem Constants */
65
66 #define NVARs      2              /* number of species          */
67 #define KH         RCONST(4.0e-6) /* horizontal diffusivity Kh */
68 #define VEL        RCONST(0.001) /* advection velocity V      */
69 #define KVO        RCONST(1.0e-8) /* coefficient in Kv(y)      */
70 #define Q1         RCONST(1.63e-16) /* coefficients q1, q2, c3 */
71 #define Q2         RCONST(4.66e-16)
72 #define C3         RCONST(3.7e16)
73 #define A3         RCONST(22.62) /* coefficient in expression for q3(t) */
74 #define A4         RCONST(7.601) /* coefficient in expression for q4(t) */
75 #define C1_SCALE   RCONST(1.0e6) /* coefficients in initial profiles */
76 #define C2_SCALE   RCONST(1.0e12)
77
78 #define T0         RCONST(0.0) /* initial time */
79 #define NOUT       12          /* number of output times */
80 #define TWOHR      RCONST(7200.0) /* number of seconds in two hours */
81 #define HALFDAY    RCONST(4.32e4) /* number of seconds in a half day */
82 #define PI         RCONST(3.1415926535898) /* pi */
83
84 #define XMIN       RCONST(0.0) /* grid boundaries in x */
85 #define XMAX       RCONST(20.0)
86 #define YMIN       RCONST(30.0) /* grid boundaries in y */
87 #define YMAX       RCONST(50.0)
88
89 #define NPEX       2           /* no. PEs in x direction of PE array */
90 #define NPEY       2           /* no. PEs in y direction of PE array */
91 /* Total no. PEs = NPEX*NPEY */
92 #define MXSUB      5           /* no. x points per subgrid */
93 #define MYSUB      5           /* no. y points per subgrid */
94
95 #define MX          (NPEX*MXSUB) /* MX = number of x mesh points */
96 #define MY          (NPEY*MYSUB) /* MY = number of y mesh points */
97 /* Spatial mesh is MX by MY */
98 /* CVodeMalloc Constants */
99
100 #define RTOL        RCONST(1.0e-5) /* scalar relative tolerance */
101 #define FLOOR        RCONST(100.0) /* value of C1 or C2 at which tolerances */
102 /* change from relative to absolute */
103 #define ATOL        (RTOL*FLOOR) /* scalar absolute tolerance */
104
105
106 /* User-defined matrix accessor macro: IJth */
107
108 /* IJth is defined in order to write code which indexes into small dense
109 matrices with a (row,column) pair, where 1 <= row,column <= NVARs.
110
111 IJth(a,i,j) references the (i,j)th entry of the small matrix realtype **a,
112 where 1 <= i,j <= NVARs. The small matrix routines in dense.h
113 work with matrices stored by column in a 2-dimensional array. In C,
114 arrays are indexed starting at 0, not 1. */
115

```

```

116 #define IJth(a,i,j) (a[j-1][i-1])
117
118 /* Type : UserData
119     contains problem constants, preconditioner blocks, pivot arrays,
120     grid constants, and processor indices */
121
122 typedef struct {
123     realtype q4, om, dx, dy, hdco, haco, vdco;
124     realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
125     int my_pe, isubx, isuby;
126     long int nvmsub, nvmsub2;
127     MPI_Comm comm;
128 } *UserData;
129
130 typedef struct {
131     void *f_data;
132     realtype **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
133     long int *pivot[MXSUB][MYSUB];
134 } *PreconData;
135
136
137 /* Private Helper Functions */
138
139 static PreconData AllocPreconData(UserData data);
140 static void InitUserData(int my_pe, MPI_Comm comm, UserData data);
141 static void FreePreconData(PreconData pdata);
142 static void SetInitialProfiles(N_Vector u, UserData data);
143 static void PrintOutput(void *cvode_mem, int my_pe, MPI_Comm comm,
144     N_Vector u, realtype t);
145 static void PrintFinalStats(void *cvode_mem);
146 static void BSend(MPI_Comm comm,
147     int my_pe, int isubx, int isuby,
148     long int dsizex, long int dsizey,
149     realtype udata[]);
150 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
151     int my_pe, int isubx, int isuby,
152     long int dsizex, long int dsizey,
153     realtype uext[], realtype buffer[]);
154 static void BRecvWait(MPI_Request request[],
155     int isubx, int isuby,
156     long int dsizex, realtype uext[],
157     realtype buffer[]);
158 static void ucomm(realtype t, N_Vector u, UserData data);
159 static void fcalc(realtype t, realtype udata[], realtype dudata[],
160     UserData data);
161
162
163 /* Functions Called by the Solver */
164
165 static int 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     boolean_t jok, boolean_t *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 = N_VARS*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 = N_VARS*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     is the relative tolerance
257     &abstol    is a pointer to the scalar absolute tolerance
258 */
259 flag = CNodeMalloc(cnode_mem, f, T0, u, CV_SS, reltol, &abstol);
260 if (check_flag(&flag, "CNodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
261
262 /* Call CVSpgrmr to specify the linear solver CVSPGMR
263     with left preconditioning and the maximum Krylov dimension maxl */
264 flag = CVSpgrmr(cnode_mem, PREC_LEFT, 0);
265 if (check_flag(&flag, "CVSpgrmr", 1, my_pe)) MPI_Abort(comm, 1);
266
267 /* Set preconditioner setup and solve routines Precond and PSolve,
268     and the pointer to the user-defined block data */
269 flag = CVSpilsSetPreconditioner(cnode_mem, Precond, PSolve, predata);
270 if (check_flag(&flag, "CVSpilsSetPreconditioner", 1, my_pe)) MPI_Abort(comm, 1);
271
272 if (my_pe == 0)
273     printf("\n2-species_adivection-diffusion_problem\n\n");
274
275 /* In loop over output points, call CNode, print results, test for error */
276 for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
277     flag = CNode(cnode_mem, tout, u, &t, CV_NORMAL);
278     if (check_flag(&flag, "CNode", 1, my_pe)) break;
279     PrintOutput(cnode_mem, my_pe, comm, u, t);
280 }
281
282 /* Print final statistics */
283 if (my_pe == 0) PrintFinalStats(cnode_mem);
284
285 /* Free memory */
286 N_VDestroy_Parallel(u);
287 free(data);
288 FreePreconData(predata);
289 CNodeFree(&cnode_mem);
290
291 MPI_Finalize();
292

```

```

293     return(0);
294 }
295
296
297 /***** Private Helper Functions *****/
298
299 /* Allocate memory for data structure of type UserData */
300
301 static PreconData AllocPreconData(UserData fdata)
302 {
303     int lx, ly;
304     PreconData pdata;
305
306     pdata = (PreconData) malloc(sizeof *pdata);
307
308     pdata->f_data = fdata;
309
310     for (lx = 0; lx < MXSUB; lx++) {
311         for (ly = 0; ly < MYSUB; ly++) {
312             (pdata->P)[lx][ly] = denalloc(NVARS);
313             (pdata->Jbd)[lx][ly] = denalloc(NVARS);
314             (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
315         }
316     }
317
318     return(pdata);
319 }
320
321 /* Load constants in data */
322
323 static void InitUserData(int my_pe, MPI_Comm comm, UserData data)
324 {
325     int isubx, isuby;
326
327     /* Set problem constants */
328     data->om = PI/HALFDAY;
329     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
330     data->dy = (YMAX-YMIN)/((realtype)(MY-1));
331     data->hdco = KH/SQR(data->dx);
332     data->haco = VEL/(RCONST(2.0)*data->dx);
333     data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;
334
335     /* Set machine-related constants */
336     data->comm = comm;
337     data->my_pe = my_pe;
338
339     /* isubx and isuby are the PE grid indices corresponding to my_pe */
340     isuby = my_pe/NPEX;
341     isubx = my_pe - isuby*NPEX;
342     data->isubx = isubx;
343     data->isuby = isuby;
344
345     /* Set the sizes of a boundary x-line in u and uext */
346     data->nvmxsub = NVARS*MXSUB;
347     data->nvmxsub2 = NVARS*(MXSUB+2);
348 }
349
350 /* Free preconditioner data memory */
351

```

```

352 static void FreePreconData(PreconData pdata)
353 {
354     int lx, ly;
355
356     for (lx = 0; lx < MXSUB; lx++) {
357         for (ly = 0; ly < MYSUB; ly++) {
358             denfree((pdata->P)[lx][ly]);
359             denfree((pdata->Jbd)[lx][ly]);
360             denfreepiv((pdata->pivot)[lx][ly]);
361         }
362     }
363
364     free(pdata);
365 }
366
367 /* Set initial conditions in u */
368
369 static void SetInitialProfiles(N_Vector u, UserData data)
370 {
371     int isubx, isuby, lx, ly, jx, jy;
372     long int offset;
373     realtype dx, dy, x, y, cx, cy, xmid, ymid;
374     realtype *udata;
375
376     /* Set pointer to data array in vector u */
377     udata = NV_DATA_P(u);
378
379     /* Get mesh spacings, and subgrid indices for this PE */
380     dx = data->dx;      dy = data->dy;
381     isubx = data->isubx; isuby = data->isuby;
382
383     /* Load initial profiles of c1 and c2 into local u vector.
384     Here lx and ly are local mesh point indices on the local subgrid,
385     and jx and jy are the global mesh point indices. */
386     offset = 0;
387     xmid = RCONST(0.5)*(XMIN + XMAX);
388     ymid = RCONST(0.5)*(YMIN + YMAX);
389     for (ly = 0; ly < MYSUB; ly++) {
390         jy = ly + isuby*MYSUB;
391         y = YMIN + jy*dy;
392         cy = SQR(RCONST(0.1)*(y - ymid));
393         cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
394         for (lx = 0; lx < MXSUB; lx++) {
395             jx = lx + isubx*MXSUB;
396             x = XMIN + jx*dx;
397             cx = SQR(RCONST(0.1)*(x - xmid));
398             cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
399             udata[offset] = C1_SCALE*cx*cy;
400             udata[offset+1] = C2_SCALE*cx*cy;
401             offset = offset + 2;
402         }
403     }
404 }
405
406 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
407
408 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
409                        N_Vector u, realtype t)
410 {

```

```

411     int qu, flag;
412     realtype hu, *udata, tempu[2];
413     int npelast;
414     long int i0, i1, nst;
415     MPI_Status status;
416
417     npelast = NPEX*NPEY - 1;
418     udata = NV_DATA_P(u);
419
420     /* Send c1,c2 at top right mesh point to PE 0 */
421     if (my_pe == npelast) {
422         i0 = NVAR*MXSUB*MYSUB - 2;
423         i1 = i0 + 1;
424         if (npelast != 0)
425             MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
426         else {
427             tempu[0] = udata[i0];
428             tempu[1] = udata[i1];
429         }
430     }
431
432     /* On PE 0, receive c1,c2 at top right, then print performance data
433        and sampled solution values */
434     if (my_pe == 0) {
435         if (npelast != 0)
436             MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
437         flag = CVodeGetNumSteps(cvode_mem, &nst);
438         check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
439         flag = CVodeGetLastOrder(cvode_mem, &qu);
440         check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
441         flag = CVodeGetLastStep(cvode_mem, &hu);
442         check_flag(&flag, "CVodeGetLastStep", 1, my_pe);
443
444         #if defined(SUNDIALS_EXTENDED_PRECISION)
445         printf("t=%%.2Le%%no. steps=%%ld%%order=%%d%%stepsize=%%.2Le\n",
446             t, nst, qu, hu);
447         printf("At bottom left: %c1, %c2 = %%.12.3Le%%%.12.3Le\n", udata[0], udata[1]);
448         printf("At top right: %c1, %c2 = %%.12.3Le%%%.12.3Le\n\n", tempu[0], tempu[1]);
449         #elif defined(SUNDIALS_DOUBLE_PRECISION)
450         printf("t=%%.2le%%no. steps=%%ld%%order=%%d%%stepsize=%%.2le\n",
451             t, nst, qu, hu);
452         printf("At bottom left: %c1, %c2 = %%.12.3le%%%.12.3le\n", udata[0], udata[1]);
453         printf("At top right: %c1, %c2 = %%.12.3le%%%.12.3le\n\n", tempu[0], tempu[1]);
454         #else
455         printf("t=%%.2e%%no. steps=%%ld%%order=%%d%%stepsize=%%.2e\n",
456             t, nst, qu, hu);
457         printf("At bottom left: %c1, %c2 = %%.12.3e%%%.12.3e\n", udata[0], udata[1]);
458         printf("At top right: %c1, %c2 = %%.12.3e%%%.12.3e\n\n", tempu[0], tempu[1]);
459         #endif
460     }
461 }
462
463 /* Print final statistics contained in iopt */
464
465 static void PrintFinalStats(void *cvode_mem)
466 {
467     long int lenrw, leniw ;
468     long int lenrwLS, leniwLS;
469     long int nst, nfe, nsetups, nni, ncnf, netf;

```



```

470     long int nli, npe, nps, ncfl, nfeLS;
471     int flag;
472
473     flag = CNodeGetWorkSpace(cvode_mem, &lenrw, &leniw);
474     check_flag(&flag, "CNodeGetWorkSpace", 1, 0);
475     flag = CNodeGetNumSteps(cvode_mem, &nst);
476     check_flag(&flag, "CNodeGetNumSteps", 1, 0);
477     flag = CNodeGetNumRhsEvals(cvode_mem, &nfe);
478     check_flag(&flag, "CNodeGetNumRhsEvals", 1, 0);
479     flag = CNodeGetNumLinSolvSetups(cvode_mem, &nsetups);
480     check_flag(&flag, "CNodeGetNumLinSolvSetups", 1, 0);
481     flag = CNodeGetNumErrTestFails(cvode_mem, &netf);
482     check_flag(&flag, "CNodeGetNumErrTestFails", 1, 0);
483     flag = CNodeGetNumNonlinSolvIters(cvode_mem, &nni);
484     check_flag(&flag, "CNodeGetNumNonlinSolvIters", 1, 0);
485     flag = CNodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
486     check_flag(&flag, "CNodeGetNumNonlinSolvConvFails", 1, 0);
487
488     flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
489     check_flag(&flag, "CVSpilsGetWorkSpace", 1, 0);
490     flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
491     check_flag(&flag, "CVSpilsGetNumLinIters", 1, 0);
492     flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
493     check_flag(&flag, "CVSpilsGetNumPrecEvals", 1, 0);
494     flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
495     check_flag(&flag, "CVSpilsGetNumPrecSolves", 1, 0);
496     flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
497     check_flag(&flag, "CVSpilsGetNumConvFails", 1, 0);
498     flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
499     check_flag(&flag, "CVSpilsGetNumRhsEvals", 1, 0);
500
501     printf("\nFinal_Statistics:\n\n");
502     printf("lenrw_uuu=%5lduuuuu leniw_uuu=%5ld\n", lenrw, leniw);
503     printf("lenrwls_u=%5lduuuuu leniwls_u=%5ld\n", lenrwLS, leniwLS);
504     printf("nst_uuuuuu=%5ld\n", nst);
505     printf("nfe_uuuuuu=%5lduuuuu nfeLS_uuu=%5ld\n", nfe, nfeLS);
506     printf("nni_uuuuuu=%5lduuuuu nli_uuuuuu=%5ld\n", nni, nli);
507     printf("nsetups_u=%5lduuuuu netf_uuuuuu=%5ld\n", nsetups, netf);
508     printf("npe_uuuuuu=%5lduuuuu nps_uuuuuu=%5ld\n", npe, nps);
509     printf("ncfn_uuuuuu=%5lduuuuu ncfl_uuuuuu=%5ld\n\n", ncfn, ncfl);
510 }
511
512 /* Routine to send boundary data to neighboring PEs */
513
514 static void BSend(MPI_Comm comm,
515                  int my_pe, int isubx, int isuby,
516                  long int dsizex, long int dsizey,
517                  realtype udata[])
518 {
519     int i, ly;
520     long int offsetu, offsetbuf;
521     realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
522
523     /* If isuby > 0, send data from bottom x-line of u */
524     if (isuby != 0)
525         MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
526
527     /* If isuby < NPEY-1, send data from top x-line of u */
528     if (isuby != NPEY-1) {

```

```

529     offsetu = (MYSUB-1)*dsizex;
530     MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
531 }
532
533 /* If isubx > 0, send data from left y-line of u (via bufleft) */
534 if (isubx != 0) {
535     for (ly = 0; ly < MYSUB; ly++) {
536         offsetbuf = ly*NVARs;
537         offsetu = ly*dsizex;
538         for (i = 0; i < NVARs; i++)
539             bufleft[offsetbuf+i] = udata[offsetu+i];
540     }
541     MPI_Send(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
542 }
543
544 /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
545 if (isubx != NPEX-1) {
546     for (ly = 0; ly < MYSUB; ly++) {
547         offsetbuf = ly*NVARs;
548         offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARs;
549         for (i = 0; i < NVARs; i++)
550             bufright[offsetbuf+i] = udata[offsetu+i];
551     }
552     MPI_Send(&bufright[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
553 }
554 }
555
556 /* Routine to start receiving boundary data from neighboring PEs.
557    Notes:
558    1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
559    passed to both the BRecvPost and BRecvWait functions, and should not
560    be manipulated between the two calls.
561    2) request should have 4 entries, and should be passed in both calls also. */
562
563 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
564                      int my_pe, int isubx, int isuby,
565                      long int dsizex, long int dsizex,
566                      realtype uext[], realtype buffer[])
567 {
568     long int offsetue;
569     /* Have bufleft and bufright use the same buffer */
570     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
571
572     /* If isuby > 0, receive data for bottom x-line of uext */
573     if (isuby != 0)
574         MPI_Irecv(&uext[NVARs], dsizex, PVEC_REAL_MPI_TYPE,
575                  my_pe-NPEX, 0, comm, &request[0]);
576
577     /* If isuby < NPEY-1, receive data for top x-line of uext */
578     if (isuby != NPEY-1) {
579         offsetue = NVARs*(1 + (MYSUB+1)*(MXSUB+2));
580         MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
581                  my_pe+NPEX, 0, comm, &request[1]);
582     }
583
584     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
585     if (isubx != 0) {
586         MPI_Irecv(&bufleft[0], dsizex, PVEC_REAL_MPI_TYPE,
587                  my_pe-1, 0, comm, &request[2]);

```

```

588     }
589
590     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
591     if (isubx != NPEX-1) {
592         MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
593                 my_pe+1, 0, comm, &request[3]);
594     }
595 }
596
597 /* Routine to finish receiving boundary data from neighboring PEs.
598 Notes:
599 1) buffer should be able to hold 2*NVARs*MYSUB realtype entries, should be
600 passed to both the BRecvPost and BRecvWait functions, and should not
601 be manipulated between the two calls.
602 2) request should have 4 entries, and should be passed in both calls also. */
603
604 static void BRecvWait(MPI_Request request[],
605                      int isubx, int isuby,
606                      long int dsizey, realtype uext[],
607                      realtype buffer[])
608 {
609     int i, ly;
610     long int dsizey2, offsetue, offsetbuf;
611     realtype *bufleft = buffer, *bufright = buffer+NVARs*MYSUB;
612     MPI_Status status;
613
614     dsizey2 = dsizey + 2*NVARs;
615
616     /* If isuby > 0, receive data for bottom x-line of uext */
617     if (isuby != 0)
618         MPI_Wait(&request[0], &status);
619
620     /* If isuby < NPEY-1, receive data for top x-line of uext */
621     if (isuby != NPEY-1)
622         MPI_Wait(&request[1], &status);
623
624     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
625     if (isubx != 0) {
626         MPI_Wait(&request[2], &status);
627
628         /* Copy the buffer to uext */
629         for (ly = 0; ly < MYSUB; ly++) {
630             offsetbuf = ly*NVARs;
631             offsetue = (ly+1)*dsizey2;
632             for (i = 0; i < NVARs; i++)
633                 uext[offsetue+i] = bufleft[offsetbuf+i];
634         }
635     }
636
637     /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
638     if (isubx != NPEX-1) {
639         MPI_Wait(&request[3], &status);
640
641         /* Copy the buffer to uext */
642         for (ly = 0; ly < MYSUB; ly++) {
643             offsetbuf = ly*NVARs;
644             offsetue = (ly+2)*dsizey2 - NVARs;
645             for (i = 0; i < NVARs; i++)
646                 uext[offsetue+i] = bufright[offsetbuf+i];

```

```

647     }
648 }
649 }
650
651 /* ucomm routine. This routine performs all communication
652    between processors of data needed to calculate f. */
653
654 static void ucomm(realtype t, N_Vector u, UserData data)
655 {
656
657     realtype *udata, *uext, buffer[2*NVARSMYSUB];
658     MPI_Comm comm;
659     int my_pe, isubx, isuby;
660     long int nvmsub, nvmysub;
661     MPI_Request request[4];
662
663     udata = NV_DATA_P(u);
664
665     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
666     comm = data->comm; my_pe = data->my_pe;
667     isubx = data->isubx; isuby = data->isuby;
668     nvmsub = data->nvmsub;
669     nvmysub = NVARSMYSUB;
670     uext = data->uext;
671
672     /* Start receiving boundary data from neighboring PEs */
673     BRecvPost(comm, request, my_pe, isubx, isuby, nvmsub, nvmysub, uext, buffer);
674
675     /* Send data from boundary of local grid to neighboring PEs */
676     BSend(comm, my_pe, isubx, isuby, nvmsub, nvmysub, udata);
677
678     /* Finish receiving boundary data from neighboring PEs */
679     BRecvWait(request, isubx, isuby, nvmsub, uext, buffer);
680 }
681
682 /* fcalc routine. Compute f(t,y). This routine assumes that communication
683    between processors of data needed to calculate f has already been done,
684    and this data is in the work array uext. */
685
686 static void fcalc(realtype t, realtype udata[],
687                  realtype dudata[], UserData data)
688 {
689     realtype *uext;
690     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
691     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
692     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
693     realtype q4coef, dely, verdco, hordco, horaco;
694     int i, lx, ly, jx, jy;
695     int isubx, isuby;
696     long int nvmsub, nvmsub2, offsetu, offsetue;
697
698     /* Get subgrid indices, data sizes, extended work array uext */
699     isubx = data->isubx; isuby = data->isuby;
700     nvmsub = data->nvmsub; nvmsub2 = data->nvmsub2;
701     uext = data->uext;
702
703     /* Copy local segment of u vector into the working extended array uext */
704     offsetu = 0;
705     offsetue = nvmsub2 + NVARSMYSUB;

```

```

706 for (ly = 0; ly < MYSUB; ly++) {
707     for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
708     offsetu = offsetu + nvmsub;
709     offsetue = offsetue + nvmsub2;
710 }
711
712 /* To facilitate homogeneous Neumann boundary conditions, when this is
713 a boundary PE, copy data from the first interior mesh line of u to uext */
714
715 /* If isuby = 0, copy x-line 2 of u to uext */
716 if (isuby == 0) {
717     for (i = 0; i < nvmsub; i++) uext[NVARS+i] = udata[nvmsub+i];
718 }
719
720 /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
721 if (isuby == NPEY-1) {
722     offsetu = (MYSUB-2)*nvmsub;
723     offsetue = (MYSUB+1)*nvmsub2 + NVARS;
724     for (i = 0; i < nvmsub; i++) uext[offsetue+i] = udata[offsetu+i];
725 }
726
727 /* If isubx = 0, copy y-line 2 of u to uext */
728 if (isubx == 0) {
729     for (ly = 0; ly < MYSUB; ly++) {
730         offsetu = ly*nvmsub + NVARS;
731         offsetue = (ly+1)*nvmsub2;
732         for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
733     }
734 }
735
736 /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
737 if (isubx == NPEX-1) {
738     for (ly = 0; ly < MYSUB; ly++) {
739         offsetu = (ly+1)*nvmsub - 2*NVARS;
740         offsetue = (ly+2)*nvmsub2 - NVARS;
741         for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
742     }
743 }
744
745 /* Make local copies of problem variables, for efficiency */
746 dely = data->dy;
747 verdco = data->vdco;
748 hordco = data->hdco;
749 horaco = data->haco;
750
751 /* Set diurnal rate coefficients as functions of t, and save q4 in
752 data block for use by preconditioner evaluation routine */
753 s = sin((data->om)*t);
754 if (s > RCONST(0.0)) {
755     q3 = EXP(-A3/s);
756     q4coef = EXP(-A4/s);
757 } else {
758     q3 = RCONST(0.0);
759     q4coef = RCONST(0.0);
760 }
761 data->q4 = q4coef;
762
763 /* Loop over all grid points in local subgrid */
764 for (ly = 0; ly < MYSUB; ly++) {

```

```

765
766     jy = ly + isuby*MYSUB;
767
768     /* Set vertical diffusion coefficients at jy +/- 1/2 */
769     ydn = YMIN + (jy - RCONST(0.5))*dely;
770     yup = ydn + dely;
771     cydn = verdco*EXP(RCONST(0.2)*ydn);
772     cyup = verdco*EXP(RCONST(0.2)*yup);
773     for (lx = 0; lx < MXSUB; lx++) {
774
775         jx = lx + isubx*MXSUB;
776
777         /* Extract c1 and c2, and set kinetic rate terms */
778         offsetue = (lx+1)*NVAR + (ly+1)*nvmsub2;
779         c1 = uext[offsetue];
780         c2 = uext[offsetue+1];
781         qq1 = Q1*c1*C3;
782         qq2 = Q2*c1*c2;
783         qq3 = q3*C3;
784         qq4 = q4coef*c2;
785         rkin1 = -qq1 - qq2 + RCONST(2.0)*qq3 + qq4;
786         rkin2 = qq1 - qq2 - qq4;
787
788         /* Set vertical diffusion terms */
789         c1dn = uext[offsetue-nvmsub2];
790         c2dn = uext[offsetue-nvmsub2+1];
791         c1up = uext[offsetue+nvmsub2];
792         c2up = uext[offsetue+nvmsub2+1];
793         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
794         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
795
796         /* Set horizontal diffusion and advection terms */
797         c1lt = uext[offsetue-2];
798         c2lt = uext[offsetue-1];
799         c1rt = uext[offsetue+2];
800         c2rt = uext[offsetue+3];
801         hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
802         hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
803         horad1 = horaco*(c1rt - c1lt);
804         horad2 = horaco*(c2rt - c2lt);
805
806         /* Load all terms into dudata */
807         offsetu = lx*NVAR + ly*nvmsub;
808         dudata[offsetu] = vertd1 + hord1 + horad1 + rkin1;
809         dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
810     }
811 }
812 }
813
814
815 /****** Functions Called by the Solver *****/
816
817 /* f routine. Evaluate f(t,y). First call ucomm to do communication of
818 subgrid boundary data into uext. Then calculate f by a call to fcalc. */
819
820 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
821 {
822     realtype *udata, *dudata;
823     UserData data;

```

```

824
825     udata = NV_DATA_P(u);
826     dudata = NV_DATA_P(udot);
827     data = (UserData) f_data;
828
829     /* Call ucomm to do inter-processor communication */
830     ucomm(t, u, data);
831
832     /* Call fcalc to calculate all right-hand sides */
833     fcalc(t, udata, dudata, data);
834
835     return(0);
836 }
837
838 /* Preconditioner setup routine. Generate and preprocess P. */
839 static int Precond(realtype tn, N_Vector u, N_Vector fu,
840                  booleantype jok, booleantype *jcurPtr,
841                  realtype gamma, void *P_data,
842                  N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
843 {
844     realtype c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
845     realtype **(*P)[MYSUB], **(*Jbd)[MYSUB];
846     long int nvmxsub, (*pivot)[MYSUB], ier, offset;
847     int lx, ly, jx, jy, isubx, isuby;
848     realtype *udata, **a, **j;
849     PreconData predata;
850     UserData data;
851
852     /* Make local copies of pointers in P_data, pointer to u's data,
853        and PE index pair */
854     predata = (PreconData) P_data;
855     data = (UserData) (predata->f_data);
856     P = predata->P;
857     Jbd = predata->Jbd;
858     pivot = predata->pivot;
859     udata = NV_DATA_P(u);
860     isubx = data->isubx;    isuby = data->isuby;
861     nvmxsub = data->nvmxsub;
862
863     if (jok) {
864
865         /* jok = TRUE: Copy Jbd to P */
866         for (ly = 0; ly < MYSUB; ly++)
867             for (lx = 0; lx < MXSUB; lx++)
868                 dencopy(Jbd[lx][ly], P[lx][ly], NVARs);
869
870         *jcurPtr = FALSE;
871     }
872
873     else {
874
875         /* jok = FALSE: Generate Jbd from scratch and copy to P */
876
877         /* Make local copies of problem variables, for efficiency */
878         q4coef = data->q4;
879         dely = data->dy;
880         verdco = data->vdco;
881         hordco = data->hdco;

```

```

883
884 /* Compute 2x2 diagonal Jacobian blocks (using q4 values
885    computed on the last f call). Load into P. */
886 for (ly = 0; ly < MYSUB; ly++) {
887     jy = ly + isuby*MYSUB;
888     ydn = YMIN + (jy - RCONST(0.5))*dely;
889     yup = ydn + dely;
890     cydn = verdco*EXP(RCONST(0.2)*ydn);
891     cyup = verdco*EXP(RCONST(0.2)*yup);
892     diag = -(cydn + cyup + RCONST(2.0)*hordco);
893     for (lx = 0; lx < MXSUB; lx++) {
894         jx = lx + isubx*MXSUB;
895         offset = lx*NVARs + ly*nvmxsub;
896         c1 = udata[offset];
897         c2 = udata[offset+1];
898         j = Jbd[lx][ly];
899         a = P[lx][ly];
900         IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
901         IJth(j,1,2) = -Q2*c1 + q4coef;
902         IJth(j,2,1) = Q1*C3 - Q2*c2;
903         IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
904         dencopy(j, a, NVARS);
905     }
906 }
907
908 *jcurPtr = TRUE;
909
910 }
911
912 /* Scale by -gamma */
913 for (ly = 0; ly < MYSUB; ly++)
914     for (lx = 0; lx < MXSUB; lx++)
915         denscale(-gamma, P[lx][ly], NVARS);
916
917 /* Add identity matrix and do LU decompositions on blocks in place */
918 for (lx = 0; lx < MXSUB; lx++) {
919     for (ly = 0; ly < MYSUB; ly++) {
920         denaddI(P[lx][ly], NVARS);
921         ier = gefa(P[lx][ly], NVARS, pivot[lx][ly]);
922         if (ier != 0) return(1);
923     }
924 }
925
926 return(0);
927 }
928
929 /* Preconditioner solve routine */
930 static int PSolve(realtype tn, N_Vector u, N_Vector fu,
931                  N_Vector r, N_Vector z,
932                  realtype gamma, realtype delta,
933                  int lr, void *P_data, N_Vector vtemp)
934 {
935     realtype **(*P)[MYSUB];
936     long int nvmxsub, *(*pivot)[MYSUB];
937     int lx, ly;
938     realtype *zdata, *v;
939     PreconData predata;
940     UserData data;
941

```



```

942  /* Extract the P and pivot arrays from P_data */
943  predata = (PreconData) P_data;
944  data = (UserData) (predata->f_data);
945  P = predata->P;
946  pivot = predata->pivot;
947
948  /* Solve the block-diagonal system Px = r using LU factors stored
949     in P and pivot data in pivot, and return the solution in z.
950     First copy vector r to z. */
951  N_VScale(RCONST(1.0), r, z);
952
953  nvmsub = data->nvmsub;
954  zdata = NV_DATA_P(z);
955
956  for (lx = 0; lx < MXSUB; lx++) {
957      for (ly = 0; ly < MYSUB; ly++) {
958          v = &(zdata[lx*NVAR + ly*nvmsub]);
959          gesl(P[lx][ly], NVAR, pivot[lx][ly], v);
960      }
961  }
962
963  return(0);
964 }
965
966
967 /***** Private Helper Function *****/
968
969 /* Check function return value...
970    opt == 0 means SUNDIALS function allocates memory so check if
971        returned NULL pointer
972    opt == 1 means SUNDIALS function returns a flag so check if
973        flag >= 0
974    opt == 2 means function allocates memory so check if returned
975        NULL pointer */
976
977 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
978 {
979     int *errflag;
980
981     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
982     if (opt == 0 && flagvalue == NULL) {
983         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
984             id, funcname);
985         return(1); }
986
987     /* Check if flag < 0 */
988     else if (opt == 1) {
989         errflag = (int *) flagvalue;
990         if (*errflag < 0) {
991             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
992                 id, funcname, *errflag);
993             return(1); }}
994
995     /* Check if function returned NULL pointer - no memory allocated */
996     else if (opt == 2 && flagvalue == NULL) {
997         fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
998             id, funcname);
999         return(1); }
1000

```

```
1001     return(0);  
1002 }
```

F Listing of cvkryx_bbd_p.c

```

1  /*
2  * -----
3  * $Revision: 1.6 $
4  * $Date: 2006/03/23 01:21:38 $
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

```

```

58 #include "ccode.h"          /* prototypes for CVODE fcts. */
59 #include "nvector_parallel.h" /* definition N_Vector and macro NV_DATA_P */
60 #include "ccode_spgmr.h"     /* prototypes and constants for CVSPGMR solver */
61 #include "ccode_bbdpre.h"    /* prototypes for CVBBDPRE module */
62 #include "sundials_types.h"  /* definitions of realtype, booleantype */
63 #include "sundials_math.h"   /* definition of macros SQR and EXP */
64
65 #include "mpi.h"             /* MPI constants and types */
66
67
68 /* Problem Constants */
69
70 #define ZERO                RCONST(0.0)
71
72 #define NVARs                2          /* number of species          */
73 #define KH                   RCONST(4.0e-6) /* horizontal diffusivity Kh */
74 #define VEL                   RCONST(0.001) /* advection velocity V      */
75 #define KVO                   RCONST(1.0e-8) /* coefficient in Kv(y)      */
76 #define Q1                   RCONST(1.63e-16) /* coefficients q1, q2, c3   */
77 #define Q2                   RCONST(4.66e-16)
78 #define C3                   RCONST(3.7e16)
79 #define A3                   RCONST(22.62) /* coefficient in expression for q3(t) */
80 #define A4                   RCONST(7.601) /* coefficient in expression for q4(t) */
81 #define C1_SCALE             RCONST(1.0e6) /* coefficients in initial profiles */
82 /*
83 #define C2_SCALE             RCONST(1.0e12)
84
85 #define T0                   ZERO          /* initial time */
86 #define NOUT                  12          /* number of output times */
87 #define TWOHR                 RCONST(7200.0) /* number of seconds in two hours */
88 #define HALFDAY               RCONST(4.32e4) /* number of seconds in a half day */
89 #define PI                    RCONST(3.1415926535898) /* pi */
90
91 #define XMIN                  ZERO          /* grid boundaries in x */
92 #define XMAX                  RCONST(20.0)
93 #define YMIN                  RCONST(30.0) /* grid boundaries in y */
94 #define YMAX                  RCONST(50.0)
95
96 #define NPEX                  2          /* no. PEs in x direction of PE array */
97 #define NPEY                  2          /* no. PEs in y direction of PE array */
98 /* Total no. PEs = NPEX*NPEY */
99 #define MXSUB                 5          /* no. x points per subgrid */
100 #define MYSUB                 5          /* no. y points per subgrid */
101
102 #define MX                    (NPEX*MXSUB) /* MX = number of x mesh points */
103 #define MY                    (NPEY*MYSUB) /* MY = number of y mesh points */
104 /* Spatial mesh is MX by MY */
105
106 /* CVodeMalloc Constants */
107
108 #define RTOL                   RCONST(1.0e-5) /* scalar relative tolerance */
109 #define FLOOR                 RCONST(100.0) /* value of C1 or C2 at which tolerances */
110 /* change from relative to absolute */
111 #define ATOL                   (RTOL*FLOOR) /* scalar absolute tolerance */
112
113 /* Type : UserData
114 contains problem constants, extended dependent variable array,
115 grid constants, processor indices, MPI communicator */
116
117 typedef struct {

```

```

116     realtype q4, om, dx, dy, hdco, haco, vdco;
117     realtype uext[NVARS*(MXSUB+2)*(MYSUB+2)];
118     int my_pe, isubx, isuby;
119     long int nvmxsub, nvmxsub2, Nlocal;
120     MPI_Comm comm;
121 } *UserData;
122
123 /* Prototypes of private helper functions */
124
125 static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
126                          UserData data);
127 static void SetInitialProfiles(N_Vector u, UserData data);
128 static void PrintIntro(int npes, long int mudq, long int mldq,
129                        long int mukeep, long int mlkeep);
130 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
131                         N_Vector u, realtype t);
132 static void PrintFinalStats(void *cnode_mem, void *pdata);
133 static void BSend(MPI_Comm comm,
134                  int my_pe, int isubx, int isuby,
135                  long int dsize, long int dsizey,
136                  realtype uarray[]);
137 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
138                      int my_pe, int isubx, int isuby,
139                      long int dsize, long int dsizey,
140                      realtype uext[], realtype buffer[]);
141 static void BRecvWait(MPI_Request request[],
142                      int isubx, int isuby,
143                      long int dsize, realtype uext[],
144                      realtype buffer[]);
145
146 static void fucomm(realtype t, N_Vector u, void *f_data);
147
148 /* Prototype of function called by the solver */
149
150 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data);
151
152 /* Prototype of functions called by the CVBBDPRE module */
153
154 static int flocal(long int Nlocal, realtype t, N_Vector u,
155                  N_Vector udot, void *f_data);
156
157 /* Private function to check function return values */
158
159 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
160
161 /***** Main Program *****/
162
163 int main(int argc, char *argv[])
164 {
165     UserData data;
166     void *cnode_mem;
167     void *pdata;
168     realtype abstol, reltol, t, tout;
169     N_Vector u;
170     int iout, my_pe, npes, flag, jpre;
171     long int neq, local_N, mudq, mldq, mukeep, mlkeep;
172     MPI_Comm comm;
173
174     data = NULL;

```

```

175     cvode_mem = pdata = NULL;
176     u = NULL;
177
178     /* Set problem size neq */
179     neq = NVAR*MX*MY;
180
181     /* Get processor number and total number of pe's */
182     MPI_Init(&argc, &argv);
183     comm = MPI_COMM_WORLD;
184     MPI_Comm_size(comm, &npes);
185     MPI_Comm_rank(comm, &my_pe);
186
187     if (npes != NPEX*NPEY) {
188         if (my_pe == 0)
189             fprintf(stderr, "\nMPI_ERROR(0): npes=%d is not equal to NPEX*NPEY=%d\n\n",
190                     npes, NPEX*NPEY);
191         MPI_Finalize();
192         return(1);
193     }
194
195     /* Set local length */
196     local_N = NVAR*MXSUB*MYSUB;
197
198     /* Allocate and load user data block */
199     data = (UserData) malloc(sizeof *data);
200     if(check_flag((void *)data, "malloc", 2, my_pe)) MPI_Abort(comm, 1);
201     InitUserData(my_pe, local_N, comm, data);
202
203     /* Allocate and initialize u, and set tolerances */
204     u = N_VNew_Parallel(comm, local_N, neq);
205     if(check_flag((void *)u, "N_VNew_Parallel", 0, my_pe)) MPI_Abort(comm, 1);
206     SetInitialProfiles(u, data);
207     abstol = ATOL;
208     reltol = RTOL;
209
210     /*
211      Call CVodeCreate to create the solver memory:
212
213      CV_BDF      specifies the Backward Differentiation Formula
214      CV_NEWTON   specifies a Newton iteration
215
216      A pointer to the integrator memory is returned and stored in cvode_mem.
217     */
218
219     cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
220     if(check_flag((void *)cvode_mem, "CVodeCreate", 0, my_pe)) MPI_Abort(comm, 1);
221
222     /* Set the pointer to user-defined data */
223     flag = CVodeSetFdata(cvode_mem, data);
224     if(check_flag(&flag, "CVodeSetFdata", 1, my_pe)) MPI_Abort(comm, 1);
225
226     /*
227      Call CVodeMalloc to initialize the integrator memory:
228
229      cvode_mem is the pointer to the integrator memory returned by CVodeCreate
230      f          is the user's right hand side function in y'=f(t,y)
231      T0         is the initial time
232      u          is the initial dependent variable vector
233      CV_SS      specifies scalar relative and absolute tolerances

```

```

234     reltol  is the relative tolerance
235     &abstol is a pointer to the scalar absolute tolerance
236 */
237
238 flag = CVodeMalloc(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
239 if(check_flag(&flag, "CVodeMalloc", 1, my_pe)) MPI_Abort(comm, 1);
240
241 /* Allocate preconditioner block */
242 mudq = mldq = NVAR*MXSUB;
243 mukeep = mlkeep = NVAR;
244 pdata = CVBBDPrecAlloc(cvode_mem, local_N, mudq, mldq,
245                        mukeep, mlkeep, ZERO, flocal, NULL);
246 if(check_flag((void *)pdata, "CVBBDPrecAlloc", 0, my_pe)) MPI_Abort(comm, 1);
247
248 /* Call CVBBDSPgmr to specify the linear solver CVSPGMR using the
249    CVBBDPRE preconditioner, with left preconditioning and the
250    default maximum Krylov dimension maxl */
251 flag = CVBBDSPgmr(cvode_mem, PREC_LEFT, 0, pdata);
252 if(check_flag(&flag, "CVBBDSPgmr", 1, my_pe)) MPI_Abort(comm, 1);
253
254 /* Print heading */
255 if (my_pe == 0) PrintIntro(npes, mudq, mldq, mukeep, mlkeep);
256
257 /* Loop over jpre (= PREC_LEFT, PREC_RIGHT), and solve the problem */
258 for (jpre = PREC_LEFT; jpre <= PREC_RIGHT; jpre++) {
259
260     /* On second run, re-initialize u, the integrator, CVBBDPRE, and CVSPGMR */
261
262     if (jpre == PREC_RIGHT) {
263
264         SetInitialProfiles(u, data);
265
266         flag = CVodeReInit(cvode_mem, f, T0, u, CV_SS, reltol, &abstol);
267         if(check_flag(&flag, "CVodeReInit", 1, my_pe)) MPI_Abort(comm, 1);
268
269         flag = CVBBDPrecReInit(pdata, mudq, mldq, ZERO, flocal, NULL);
270         if(check_flag(&flag, "CVBBDPrecReInit", 1, my_pe)) MPI_Abort(comm, 1);
271
272         flag = CVSpilsSetPrecType(cvode_mem, PREC_RIGHT);
273         check_flag(&flag, "CVSpilsSetPrecType", 1, my_pe);
274
275         if (my_pe == 0) {
276             printf("\n\n-----");
277             printf("-----\n");
278         }
279     }
280 }
281
282
283 if (my_pe == 0) {
284     printf("\n\nPreconditioner type is: %s\n\n",
285           (jpre == PREC_LEFT) ? "PREC_LEFT" : "PREC_RIGHT");
286 }
287
288 /* In loop over output points, call CVode, print results, test for error */
289
290 for (iout = 1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
291     flag = CVode(cvode_mem, tout, u, &t, CV_NORMAL);
292     if(check_flag(&flag, "CVode", 1, my_pe)) break;

```

```

293     PrintOutput(cvode_mem, my_pe, comm, u, t);
294 }
295
296 /* Print final statistics */
297
298 if (my_pe == 0) PrintFinalStats(cvode_mem, pdata);
299
300 } /* End of jpre loop */
301
302 /* Free memory */
303 N_VDestroy_Parallel(u);
304 CVBBDPrecFree(&pdata);
305 free(data);
306 CVodeFree(&cvode_mem);
307
308 MPI_Finalize();
309
310 return(0);
311 }
312
313 /****** Private Helper Functions *****/
314
315 /* Load constants in data */
316
317 static void InitUserData(int my_pe, long int local_N, MPI_Comm comm,
318                          UserData data)
319 {
320     int isubx, isuby;
321
322     /* Set problem constants */
323     data->om = PI/HALFDAY;
324     data->dx = (XMAX-XMIN)/((realtype)(MX-1));
325     data->dy = (YMAX-YMIN)/((realtype)(MY-1));
326     data->hdco = KH/SQR(data->dx);
327     data->haco = VEL/(RCONST(2.0)*data->dx);
328     data->vdco = (RCONST(1.0)/SQR(data->dy))*KV0;
329
330     /* Set machine-related constants */
331     data->comm = comm;
332     data->my_pe = my_pe;
333     data->Nlocal = local_N;
334     /* isubx and isuby are the PE grid indices corresponding to my_pe */
335     isuby = my_pe/NPEX;
336     isubx = my_pe - isuby*NPEX;
337     data->isubx = isubx;
338     data->isuby = isuby;
339     /* Set the sizes of a boundary x-line in u and uext */
340     data->nvmxsub = NVAR*MXSUB;
341     data->nvmxsub2 = NVAR*(MXSUB+2);
342 }
343
344 /* Set initial conditions in u */
345
346 static void SetInitialProfiles(N_Vector u, UserData data)
347 {
348     int isubx, isuby;
349     int lx, ly, jx, jy;
350     long int offset;
351     realtype dx, dy, x, y, cx, cy, xmid, ymid;

```



```

352     realtype *uarray;
353
354     /* Set pointer to data array in vector u */
355
356     uarray = NV_DATA_P(u);
357
358     /* Get mesh spacings, and subgrid indices for this PE */
359
360     dx = data->dx;          dy = data->dy;
361     isubx = data->isubx;    isuby = data->isuby;
362
363     /* Load initial profiles of c1 and c2 into local u vector.
364     Here lx and ly are local mesh point indices on the local subgrid,
365     and jx and jy are the global mesh point indices. */
366
367     offset = 0;
368     xmid = RCONST(0.5)*(XMIN + XMAX);
369     ymid = RCONST(0.5)*(YMIN + YMAX);
370     for (ly = 0; ly < MYSUB; ly++) {
371         jy = ly + isuby*MYSUB;
372         y = YMIN + jy*dy;
373         cy = SQR(RCONST(0.1)*(y - ymid));
374         cy = RCONST(1.0) - cy + RCONST(0.5)*SQR(cy);
375         for (lx = 0; lx < MXSUB; lx++) {
376             jx = lx + isubx*MXSUB;
377             x = XMIN + jx*dx;
378             cx = SQR(RCONST(0.1)*(x - xmid));
379             cx = RCONST(1.0) - cx + RCONST(0.5)*SQR(cx);
380             uarray[offset] = C1_SCALE*cx*cy;
381             uarray[offset+1] = C2_SCALE*cx*cy;
382             offset = offset + 2;
383         }
384     }
385 }
386
387 /* Print problem introduction */
388
389 static void PrintIntro(int npes, long int mudq, long int mldq,
390                       long int mukeep, long int mlkeep)
391 {
392     printf("\n2-species diurnal advection-diffusion problem\n");
393     printf("  %d by %d mesh on %d processors\n", MX, MY, npes);
394     printf("  Using CVBBDPRE preconditioner module\n");
395     printf("  Difference-quotient half-bandwidths are");
396     printf(" mudq = %ld, mldq = %ld\n", mudq, mldq);
397     printf("  Retained band block half-bandwidths are");
398     printf(" mukeep = %ld, mlkeep = %ld", mukeep, mlkeep);
399
400     return;
401 }
402
403 /* Print current t, step count, order, stepsize, and sampled c1,c2 values */
404
405 static void PrintOutput(void *cnode_mem, int my_pe, MPI_Comm comm,
406                       N_Vector u, realtype t)
407 {
408     int qu, flag, npelast;
409     long int i0, i1, nst;
410     realtype hu, *uarray, tempu[2];

```

```

411     MPI_Status status;
412
413     npelast = NPEX*NPEY - 1;
414     uarray = NV_DATA_P(u);
415
416     /* Send c1,c2 at top right mesh point to PE 0 */
417     if (my_pe == npelast) {
418         i0 = NVAR*MXSUB*MYSUB - 2;
419         i1 = i0 + 1;
420         if (npelast != 0)
421             MPI_Send(&uarray[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
422         else {
423             tempu[0] = uarray[i0];
424             tempu[1] = uarray[i1];
425         }
426     }
427
428     /* On PE 0, receive c1,c2 at top right, then print performance data
429        and sampled solution values */
430     if (my_pe == 0) {
431         if (npelast != 0)
432             MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
433         flag = CVodeGetNumSteps(cvode_mem, &nst);
434         check_flag(&flag, "CVodeGetNumSteps", 1, my_pe);
435         flag = CVodeGetLastOrder(cvode_mem, &qu);
436         check_flag(&flag, "CVodeGetLastOrder", 1, my_pe);
437         flag = CVodeGetLastStep(cvode_mem, &hu);
438         check_flag(&flag, "CVodeGetLastStep", 1, my_pe);
439         #if defined(SUNDIALS_EXTENDED_PRECISION)
440             printf("t=%%.2Le%%no. steps=%ld%%order=%d%%stepsize=%%.2Le\n",
441                 t, nst, qu, hu);
442             printf("At bottom left: %c1, %c2 = %%.12.3Le%%%.12.3Le\n", uarray[0], uarray[1]);
443             printf("At top right: %c1, %c2 = %%.12.3Le%%%.12.3Le\n\n", tempu[0], tempu[1]);
444         #elif defined(SUNDIALS_DOUBLE_PRECISION)
445             printf("t=%%.2le%%no. steps=%ld%%order=%d%%stepsize=%%.2le\n",
446                 t, nst, qu, hu);
447             printf("At bottom left: %c1, %c2 = %%.12.3le%%%.12.3le\n", uarray[0], uarray[1]);
448             printf("At top right: %c1, %c2 = %%.12.3le%%%.12.3le\n\n", tempu[0], tempu[1]);
449         #else
450             printf("t=%%.2e%%no. steps=%ld%%order=%d%%stepsize=%%.2e\n",
451                 t, nst, qu, hu);
452             printf("At bottom left: %c1, %c2 = %%.12.3e%%%.12.3e\n", uarray[0], uarray[1]);
453             printf("At top right: %c1, %c2 = %%.12.3e%%%.12.3e\n\n", tempu[0], tempu[1]);
454         #endif
455     }
456 }
457
458 /* Print final statistics contained in iopt */
459
460 static void PrintFinalStats(void *cvode_mem, void *pdata)
461 {
462     long int lenrw, leniw ;
463     long int lenrwLS, leniwLS;
464     long int lenrwBBDP, leniwBBDP, ngevalsBBDP;
465     long int nst, nfe, nsetups, nni, ncfn, netf;
466     long int nli, npe, nps, ncfl, nfeLS;
467     int flag;
468
469     flag = CVodeGetWorkspace(cvode_mem, &lenrw, &leniw);

```

```

470     check_flag(&flag, "CVodeGetWorkSpace", 1, 0);
471     flag = CVodeGetNumSteps(cvode_mem, &nst);
472     check_flag(&flag, "CVodeGetNumSteps", 1, 0);
473     flag = CVodeGetNumRhsEvals(cvode_mem, &nfe);
474     check_flag(&flag, "CVodeGetNumRhsEvals", 1, 0);
475     flag = CVodeGetNumLinSolvSetups(cvode_mem, &nsetups);
476     check_flag(&flag, "CVodeGetNumLinSolvSetups", 1, 0);
477     flag = CVodeGetNumErrTestFails(cvode_mem, &netf);
478     check_flag(&flag, "CVodeGetNumErrTestFails", 1, 0);
479     flag = CVodeGetNumNonlinSolvIters(cvode_mem, &nni);
480     check_flag(&flag, "CVodeGetNumNonlinSolvIters", 1, 0);
481     flag = CVodeGetNumNonlinSolvConvFails(cvode_mem, &ncfn);
482     check_flag(&flag, "CVodeGetNumNonlinSolvConvFails", 1, 0);
483
484     flag = CVSpilsGetWorkSpace(cvode_mem, &lenrwLS, &leniwLS);
485     check_flag(&flag, "CVSpilsGetWorkSpace", 1, 0);
486     flag = CVSpilsGetNumLinIters(cvode_mem, &nli);
487     check_flag(&flag, "CVSpilsGetNumLinIters", 1, 0);
488     flag = CVSpilsGetNumPrecEvals(cvode_mem, &npe);
489     check_flag(&flag, "CVSpilsGetNumPrecEvals", 1, 0);
490     flag = CVSpilsGetNumPrecSolves(cvode_mem, &nps);
491     check_flag(&flag, "CVSpilsGetNumPrecSolves", 1, 0);
492     flag = CVSpilsGetNumConvFails(cvode_mem, &ncfl);
493     check_flag(&flag, "CVSpilsGetNumConvFails", 1, 0);
494     flag = CVSpilsGetNumRhsEvals(cvode_mem, &nfeLS);
495     check_flag(&flag, "CVSpilsGetNumRhsEvals", 1, 0);
496
497     printf("\nFinal Statistics:\n\n");
498     printf("lenrw=%5ldleniw=%5ld\n", lenrw, leniw);
499     printf("lenrwls=%5ldleniwls=%5ld\n", lenrwLS, leniwLS);
500     printf("nst=%5ld\n", nst);
501     printf("nfe=%5ldnfels=%5ld\n", nfe, nfeLS);
502     printf("nni=%5ldnli=%5ld\n", nni, nli);
503     printf("nsetups=%5ldnetf=%5ld\n", nsetups, netf);
504     printf("npe=%5ldnps=%5ld\n", npe, nps);
505     printf("ncfn=%5ldncfl=%5ld\n", ncfn, ncfl);
506
507     flag = CVBBDPrecGetWorkSpace(pdata, &lenrwBBDP, &leniwBBDP);
508     check_flag(&flag, "CVBBDPrecGetWorkSpace", 1, 0);
509     flag = CVBBDPrecGetNumGfnEvals(pdata, &ngevalsBBDP);
510     check_flag(&flag, "CVBBDPrecGetNumGfnEvals", 1, 0);
511     printf("In CVBBDPRE: real/integer local work space sizes = %ld, %ld\n",
512           lenrwBBDP, leniwBBDP);
513     printf("no. of local evals. = %ld\n", ngevalsBBDP);
514 }
515
516 /* Routine to send boundary data to neighboring PEs */
517
518 static void BSend(MPI_Comm comm,
519                  int my_pe, int isubx, int isuby,
520                  long int dsizex, long int dsizey,
521                  realtype uarray[])
522 {
523     int i, ly;
524     long int offsetu, offsetbuf;
525     realtype bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];
526
527     /* If isuby > 0, send data from bottom x-line of u */
528

```

```

529     if (isuby != 0)
530         MPI_Send(&uarray[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
531
532     /* If isuby < NPEY-1, send data from top x-line of u */
533
534     if (isuby != NPEY-1) {
535         offsetu = (MYSUB-1)*dsizey;
536         MPI_Send(&uarray[offsetu], dsizey, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
537     }
538
539     /* If isubx > 0, send data from left y-line of u (via bufleft) */
540
541     if (isubx != 0) {
542         for (ly = 0; ly < MYSUB; ly++) {
543             offsetbuf = ly*NVAR;
544             offsetu = ly*dsizey;
545             for (i = 0; i < NVAR; i++)
546                 bufleft[offsetbuf+i] = uarray[offsetu+i];
547         }
548         MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
549     }
550
551     /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */
552
553     if (isubx != NPEX-1) {
554         for (ly = 0; ly < MYSUB; ly++) {
555             offsetbuf = ly*NVAR;
556             offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVAR;
557             for (i = 0; i < NVAR; i++)
558                 bufright[offsetbuf+i] = uarray[offsetu+i];
559         }
560         MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
561     }
562 }
563
564
565 /* Routine to start receiving boundary data from neighboring PEs.
566 Notes:
567 1) buffer should be able to hold 2*NVAR*MYSUB realtype entries, should be
568 passed to both the BRecvPost and BRecvWait functions, and should not
569 be manipulated between the two calls.
570 2) request should have 4 entries, and should be passed in both calls also. */
571
572 static void BRecvPost(MPI_Comm comm, MPI_Request request[],
573                      int my_pe, int isubx, int isuby,
574                      long int dsizey, long int dsizey,
575                      realtype uext[], realtype buffer[])
576 {
577     long int offsetue;
578     /* Have bufleft and bufright use the same buffer */
579     realtype *bufleft = buffer, *bufright = buffer+NVAR*MYSUB;
580
581     /* If isuby > 0, receive data for bottom x-line of uext */
582     if (isuby != 0)
583         MPI_Irecv(&uext[NVAR], dsizey, PVEC_REAL_MPI_TYPE,
584                  my_pe-NPEX, 0, comm, &request[0]);
585
586     /* If isuby < NPEY-1, receive data for top x-line of uext */
587     if (isuby != NPEY-1) {

```

```

588     offsetue = NVAR*(1 + (MYSUB+1)*(MXSUB+2));
589     MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
590              my_pe+NPEX, 0, comm, &request[1]);
591 }
592
593 /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
594 if (isubx != 0) {
595     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
596              my_pe-1, 0, comm, &request[2]);
597 }
598
599 /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
600 if (isubx != NPEX-1) {
601     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
602              my_pe+1, 0, comm, &request[3]);
603 }
604
605 }
606
607 /* Routine to finish receiving boundary data from neighboring PEs.
608    Notes:
609    1) buffer should be able to hold 2*NVAR*MYSUB realtype entries, should be
610    passed to both the BRecvPost and BRecvWait functions, and should not
611    be manipulated between the two calls.
612    2) request should have 4 entries, and should be passed in both calls also. */
613
614 static void BRecvWait(MPI_Request request[],
615                      int isubx, int isuby,
616                      long int dsizex, realtype uext[],
617                      realtype buffer[])
618 {
619     int i, ly;
620     long int dsizex2, offsetue, offsetbuf;
621     realtype *bufleft = buffer, *bufright = buffer+NVAR*MYSUB;
622     MPI_Status status;
623
624     dsizex2 = dsizex + 2*NVAR;
625
626     /* If isuby > 0, receive data for bottom x-line of uext */
627     if (isuby != 0)
628         MPI_Wait(&request[0], &status);
629
630     /* If isuby < NPEY-1, receive data for top x-line of uext */
631     if (isuby != NPEY-1)
632         MPI_Wait(&request[1], &status);
633
634     /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
635     if (isubx != 0) {
636         MPI_Wait(&request[2], &status);
637
638         /* Copy the buffer to uext */
639         for (ly = 0; ly < MYSUB; ly++) {
640             offsetbuf = ly*NVAR;
641             offsetue = (ly+1)*dsizex2;
642             for (i = 0; i < NVAR; i++)
643                 uext[offsetue+i] = bufleft[offsetbuf+i];
644         }
645     }
646 }

```

```

647  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
648  if (isubx != NPEX-1) {
649      MPI_Wait(&request[3], &status);
650
651      /* Copy the buffer to uext */
652      for (ly = 0; ly < MYSUB; ly++) {
653          offsetbuf = ly*NVARs;
654          offsetue = (ly+2)*dsizex2 - NVARs;
655          for (i = 0; i < NVARs; i++)
656              uext[offsetue+i] = bufright[offsetbuf+i];
657      }
658  }
659 }
660
661 /* fucomm routine. This routine performs all inter-processor
662    communication of data in u needed to calculate f. */
663
664 static void fucomm(realtype t, N_Vector u, void *f_data)
665 {
666     UserData data;
667     realtype *uarray, *uext, buffer[2*NVARs*MYSUB];
668     MPI_Comm comm;
669     int my_pe, isubx, isuby;
670     long int nvxsub, nvmysub;
671     MPI_Request request[4];
672
673     data = (UserData) f_data;
674     uarray = NV_DATA_P(u);
675
676     /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */
677
678     comm = data->comm; my_pe = data->my_pe;
679     isubx = data->isubx; isuby = data->isuby;
680     nvxsub = data->nvxsub;
681     nvmysub = NVARs*MYSUB;
682     uext = data->uext;
683
684     /* Start receiving boundary data from neighboring PEs */
685
686     BRecvPost(comm, request, my_pe, isubx, isuby, nvxsub, nvmysub, uext, buffer);
687
688     /* Send data from boundary of local grid to neighboring PEs */
689
690     BSend(comm, my_pe, isubx, isuby, nvxsub, nvmysub, uarray);
691
692     /* Finish receiving boundary data from neighboring PEs */
693
694     BRecvWait(request, isubx, isuby, nvxsub, uext, buffer);
695 }
696
697 /***** Function called by the solver *****/
698
699 /* f routine. Evaluate f(t,y). First call fucomm to do communication of
700    subgrid boundary data into uext. Then calculate f by a call to flocal. */
701
702 static int f(realtype t, N_Vector u, N_Vector udot, void *f_data)
703 {
704     UserData data;
705

```

```

706     data = (UserData) f_data;
707
708     /* Call fucomm to do inter-processor communication */
709
710     fucomm (t, u, f_data);
711
712     /* Call flocal to calculate all right-hand sides */
713
714     flocal (data->Nlocal, t, u, udot, f_data);
715
716     return(0);
717 }
718
719 /***** Functions called by the CVBBDPRE module *****/
720
721 /* flocal routine. Compute f(t,y). This routine assumes that all
722    inter-processor communication of data needed to calculate f has already
723    been done, and this data is in the work array uext. */
724
725 static int flocal(long int Nlocal, realtype t, N_Vector u,
726                  N_Vector udot, void *f_data)
727 {
728     realtype *uext;
729     realtype q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
730     realtype c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
731     realtype qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
732     realtype q4coef, dely, verdco, hordco, horaco;
733     int i, lx, ly, jx, jy;
734     int isubx, isuby;
735     long int nvmsub, nvmsub2, offsetu, offsetue;
736     UserData data;
737     realtype *uarray, *duarray;
738
739     uarray = NV_DATA_P(u);
740     duarray = NV_DATA_P(udot);
741
742     /* Get subgrid indices, array sizes, extended work array uext */
743
744     data = (UserData) f_data;
745     isubx = data->isubx; isuby = data->isuby;
746     nvmsub = data->nvmsub; nvmsub2 = data->nvmsub2;
747     uext = data->uext;
748
749     /* Copy local segment of u vector into the working extended array uext */
750
751     offsetu = 0;
752     offsetue = nvmsub2 + NVARs;
753     for (ly = 0; ly < MYSUB; ly++) {
754         for (i = 0; i < nvmsub; i++) uext[offsetue+i] = uarray[offsetu+i];
755         offsetu = offsetu + nvmsub;
756         offsetue = offsetue + nvmsub2;
757     }
758
759     /* To facilitate homogeneous Neumann boundary conditions, when this is
760        a boundary PE, copy data from the first interior mesh line of u to uext */
761
762     /* If isuby = 0, copy x-line 2 of u to uext */
763     if (isuby == 0) {
764         for (i = 0; i < nvmsub; i++) uext[NVARs+i] = uarray[nvmsub+i];

```

```

765     }
766
767     /* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uest */
768     if (isuby == NPEY-1) {
769         offsetu = (MYSUB-2)*nvmxsub;
770         offsetue = (MYSUB+1)*nvmxsub2 + NVAR;
771         for (i = 0; i < nvmxsub; i++) uest[offsetue+i] = uarray[offsetu+i];
772     }
773
774     /* If isubx = 0, copy y-line 2 of u to uest */
775     if (isubx == 0) {
776         for (ly = 0; ly < MYSUB; ly++) {
777             offsetu = ly*nvmxsub + NVAR;
778             offsetue = (ly+1)*nvmxsub2;
779             for (i = 0; i < NVAR; i++) uest[offsetue+i] = uarray[offsetu+i];
780         }
781     }
782
783     /* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uest */
784     if (isubx == NPEX-1) {
785         for (ly = 0; ly < MYSUB; ly++) {
786             offsetu = (ly+1)*nvmxsub - 2*NVAR;
787             offsetue = (ly+2)*nvmxsub2 - NVAR;
788             for (i = 0; i < NVAR; i++) uest[offsetue+i] = uarray[offsetu+i];
789         }
790     }
791
792     /* Make local copies of problem variables, for efficiency */
793
794     dely = data->dy;
795     verdco = data->vdco;
796     hordco = data->hdco;
797     horaco = data->haco;
798
799     /* Set diurnal rate coefficients as functions of t, and save q4 in
800     data block for use by preconditioner evaluation routine */
801
802     s = sin((data->om)*t);
803     if (s > ZERO) {
804         q3 = EXP(-A3/s);
805         q4coef = EXP(-A4/s);
806     } else {
807         q3 = ZERO;
808         q4coef = ZERO;
809     }
810     data->q4 = q4coef;
811
812
813     /* Loop over all grid points in local subgrid */
814
815     for (ly = 0; ly < MYSUB; ly++) {
816
817         jy = ly + isuby*MYSUB;
818
819         /* Set vertical diffusion coefficients at jy +/- 1/2 */
820
821         ydn = YMIN + (jy - RCONST(0.5))*dely;
822         yup = ydn + dely;
823         cydn = verdco*EXP(RCONST(0.2)*ydn);

```



```

824     cyup = verdco*EXP(RCONST(0.2)*yup);
825     for (lx = 0; lx < MXSUB; lx++) {
826
827         jx = lx + isubx*MXSUB;
828
829         /* Extract c1 and c2, and set kinetic rate terms */
830
831         offsetue = (lx+1)*NVARs + (ly+1)*nvmxsub2;
832         c1 = uext[offsetue];
833         c2 = uext[offsetue+1];
834         qq1 = Q1*c1*C3;
835         qq2 = Q2*c1*c2;
836         qq3 = q3*C3;
837         qq4 = q4coef*c2;
838         rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
839         rkin2 = qq1 - qq2 - qq4;
840
841         /* Set vertical diffusion terms */
842
843         c1dn = uext[offsetue-nvmxsub2];
844         c2dn = uext[offsetue-nvmxsub2+1];
845         c1up = uext[offsetue+nvmxsub2];
846         c2up = uext[offsetue+nvmxsub2+1];
847         vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
848         vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);
849
850         /* Set horizontal diffusion and advection terms */
851
852         c1lt = uext[offsetue-2];
853         c2lt = uext[offsetue-1];
854         c1rt = uext[offsetue+2];
855         c2rt = uext[offsetue+3];
856         hord1 = hordco*(c1rt - RCONST(2.0)*c1 + c1lt);
857         hord2 = hordco*(c2rt - RCONST(2.0)*c2 + c2lt);
858         horad1 = horaco*(c1rt - c1lt);
859         horad2 = horaco*(c2rt - c2lt);
860
861         /* Load all terms into duarray */
862
863         offsetu = lx*NVARs + ly*nvmxsub;
864         duarray[offsetu] = vertd1 + hord1 + horad1 + rkin1;
865         duarray[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
866     }
867 }
868
869 return(0);
870 }
871
872 /* Check function return value...
873     opt == 0 means SUNDIALS function allocates memory so check if
874         returned NULL pointer
875     opt == 1 means SUNDIALS function returns a flag so check if
876         flag >= 0
877     opt == 2 means function allocates memory so check if returned
878         NULL pointer */
879
880 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
881 {
882     int *errflag;

```

```

883
884 /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
885 if (opt == 0 && flagvalue == NULL) {
886     fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
887         id, funcname);
888     return(1); }
889
890 /* Check if flag < 0 */
891 else if (opt == 1) {
892     errflag = (int *) flagvalue;
893     if (*errflag < 0) {
894         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
895             id, funcname, *errflag);
896         return(1); }}
897
898 /* Check if function returned NULL pointer - no memory allocated */
899 else if (opt == 2 && flagvalue == NULL) {
900     fprintf(stderr, "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
901         id, funcname);
902     return(1); }
903
904 return(0);
905 }

```

G Listing of fcvkryx.f

```

1 C -----
2 C $Revision: 1.4 $
3 C $Date: 2006/03/17 16:37:33 $
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 MX and MY, for consistency with the output statements
34 C below.
35 C -----
36 C
37 C IMPLICIT NONE
38 C
39 C INTEGER*4 MX, MY, NEQ
40 C PARAMETER (MX=10, MY=10)
41 C PARAMETER (NEQ=2*MX*MY)
42 C INTEGER*4 LENIPAR, LENRPAR
43 C PARAMETER (LENIPAR=6+2*MX*MY, LENRPAR=12+8*MX*MY)
44 C
45 C INTEGER METH, ITMETH, IATOL, ITASK, IER, LNCFL, LNPS
46 C INTEGER LNST, LNFE, LNSETUP, LNNI, LNCF, LQ, LH, LNPE, LNLI, LNETF
47 C INTEGER JOUT, JPRETYPE, IGSTYPE, MAXL
48 C INTEGER*4 IOUT(25), IPAR(LenIPAR)
49 C INTEGER*4 NST, NFE, NPSET, NPE, NPS, NNI, NETF
50 C INTEGER*4 NLI, NCFN, NCFI
51 C DOUBLE PRECISION ATOL, AVDIM, T, TOUT, TWOHR, RTOL, FLOOR, DELT
52 C DOUBLE PRECISION U(2,MX,MY), ROUT(10), RPAR(LenRPAR)
53 C
54 C DATA TWOHR/7200.0D0/, RTOL/1.0D-5/, FLOOR/100.0D0/,
55 C & JPRETYPE/1/, IGSTYPE/1/, MAXL/0/, DELT/0.0D0/
56 C DATA LNST/3/, LNFE/4/, LNETF/5/, LNCF/6/, LNNI/7/, LNSETUP/8/,
57 C & LQ/9/, LNPE/18/, LNLI/20/, LNPS/19/, LNCFL/21/

```

```

58      DATA LH/2/
59  C
60  C      Load problem constants into IPAR, RPAR, and set initial values
61      CALL INITKX(MX, MY, U, IPAR, RPAR)
62  C
63  C      Set other input arguments.
64      T = 0.0D0
65      METH = 2
66      ITMETH = 2
67      IATOL = 1
68      ATOL = RTOL * FLOOR
69      ITASK = 1
70  C
71      WRITE(6,10) NEQ
72  10      FORMAT('Krylov example problem: '//
73      &          ' Kinetics-transport, NEQ = ', I4/)
74  C
75      CALL FNVINITS(1, NEQ, IER)
76      IF (IER .NE. 0) THEN
77          WRITE(6,20) IER
78  20      FORMAT('SUNDIALS_ERROR: FNVINITS returned IER = ', I5)
79          STOP
80      ENDIF
81  C
82  C      Initialize CVMODE
83      CALL FCVMALLOC(T, U, METH, ITMETH, IATOL, RTOL, ATOL,
84      &          IOUT, ROUT, IPAR, RPAR, IER)
85      IF (IER .NE. 0) THEN
86          WRITE(6,30) IER
87  30      FORMAT('SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
88          STOP
89      ENDIF
90  C
91      CALL FCVSPGMR(JPRETYPE, IGSTYPE, MAXL, DELT, IER)
92      IF (IER .NE. 0) THEN
93          WRITE(6,40) IER
94  40      FORMAT('SUNDIALS_ERROR: FCVSPGMR returned IER = ', I5)
95          CALL FCVFREE
96          STOP
97      ENDIF
98  C
99      CALL FCVSPILSSETPREC(1, IER)
100 C
101 C      Loop over output points, call FCVMODE, print sample solution values.
102      TOUT = TWOHR
103      DO JOUT = 1, 12
104  C
105          CALL FCVMODE(TOUT, T, U, ITASK, IER)
106  C
107          WRITE(6,50) T, IOUT(LNST), IOUT(LQ), ROUT(LH)
108  50      FORMAT(' t = ', E11.3, 3X, 'nst = ', I5,
109      &          ' q = ', I2, ' h = ', E14.6)
110          WRITE(6,55) U(1,1,1), U(1,5,5), U(1,10,10),
111      &          U(2,1,1), U(2,5,5), U(2,10,10)
112  55      FORMAT(' c1 (bot.left/middle/top rt.) = ', 3E14.6/
113      &          ' c2 (bot.left/middle/top rt.) = ', 3E14.6)
114  C
115          IF (IER .NE. 0) THEN
116              WRITE(6,60) IER, IOUT(15)

```

```

117 60      FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
118      &      ' , Linear Solver returned IER = ', I5)
119      CALL FCVFREE
120      STOP
121      ENDIF
122 C
123      TOUT = TOUT + TWOHR
124 C
125      ENDDO
126
127 C      Print final statistics.
128      NST = IOUT(LNST)
129      NFE = IOUT(LNFE)
130      NPSET = IOUT(LNSETUP)
131      NPE = IOUT(LNPE)
132      NPS = IOUT(LNPS)
133      NNI = IOUT(LNNI)
134      NLI = IOUT(LNLI)
135      AVDIM = DBLE(NLI) / DBLE(NNI)
136      NCFN = IOUT(LNCF)
137      NCFL = IOUT(LNCFL)
138      NETF = IOUT(LNETF)
139      WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
140      &      NCFL, NETF
141 80  FORMAT(// 'Final statistics: '//
142      &      ' number of steps = ', I5, 5X,
143      &      ' number of f evals. = ', I5/
144      &      ' number of prec. setups = ', I5/
145      &      ' number of prec. evals. = ', I5, 5X,
146      &      ' number of prec. solves = ', I5/
147      &      ' number of nonl. iters. = ', I5, 5X,
148      &      ' number of lin. iters. = ', I5/
149      &      ' average Krylov subspace dimension (NLI/NNI) = ', E14.6/
150      &      ' number of conv. failures.. nonlinear = ', I3,
151      &      ' linear = ', I3/
152      &      ' number of error test failures = ', I3)
153 C
154      CALL FCVFREE
155 C
156      STOP
157      END
158
159 C      -----
160
161      SUBROUTINE INITKX(MX, MY, U0, IPAR, RPAR)
162 C      Routine to set problem constants and initial values
163 C
164      IMPLICIT NONE
165 C
166      INTEGER*4 MX, MY, IPAR(*)
167      DOUBLE PRECISION RPAR(*)
168 C
169      INTEGER*4 MM, JY, JX, P_IPP, P_BD, P_P
170      DOUBLE PRECISION U0
171      DIMENSION U0(2,MX,MY)
172      DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
173      DOUBLE PRECISION VDCO, HACO, X, Y
174      DOUBLE PRECISION CX, CY, DKH, DKVO, DX, HALFDA, PI, VEL
175 C

```

```

176      DATA DKH/4.0D-6/, VEL/0.001D0/, DKV0/1.0D-8/, HALFDA/4.32D4/,
177      1      PI/3.1415926535898D0/
178  C
179  C      Problem constants
180      MM = MX * MY
181      Q1 = 1.63D-16
182      Q2 = 4.66D-16
183      A3 = 22.62D0
184      A4 = 7.601D0
185      OM = PI / HALFDA
186      C3 = 3.7D16
187      DX = 20.0D0 / (MX - 1.0D0)
188      DY = 20.0D0 / (MY - 1.0D0)
189      HDCO = DKH / DX**2
190      HACO = VEL / (2.0D0 * DX)
191      VDCO = (1.0D0 / DY**2) * DKV0
192  C
193  C      Load constants in IPAR and RPAR
194      IPAR(1) = MX
195      IPAR(2) = MY
196      IPAR(3) = MM
197  C
198      RPAR(1) = Q1
199      RPAR(2) = Q2
200      RPAR(3) = Q3
201      RPAR(4) = Q4
202      RPAR(5) = A3
203      RPAR(6) = A4
204      RPAR(7) = OM
205      RPAR(8) = C3
206      RPAR(9) = DY
207      RPAR(10) = HDCO
208      RPAR(11) = VDCO
209      RPAR(12) = HACO
210  C
211  C      Pointers into IPAR and RPAR
212      P_IPP = 7
213      P_BD = 13
214      P_P = P_BD + 4*MM
215  C
216      IPAR(4) = P_IPP
217      IPAR(5) = P_BD
218      IPAR(6) = P_P
219  C
220  C      Set initial profiles.
221      DO JY = 1, MY
222          Y = 30.0D0 + (JY - 1.0D0) * DY
223          CY = (0.1D0 * (Y - 40.0D0))**2
224          CY = 1.0D0 - CY + 0.5D0 * CY**2
225      DO JX = 1, MX
226          X = (JX - 1.0D0) * DX
227          CX = (0.1D0 * (X - 10.0D0))**2
228          CX = 1.0D0 - CX + 0.5D0 * CX**2
229          UO(1,JX,JY) = 1.0D6 * CX * CY
230          UO(2,JX,JY) = 1.0D12 * CX * CY
231      ENDDO
232      ENDDO
233  C
234      RETURN

```

```

235      END
236
237      C -----
238
239      SUBROUTINE FCVFUN(T, U, UDOT, IPAR, RPAR, IER)
240      C Routine for right-hand side function f
241      C
242      IMPLICIT NONE
243      C
244      DOUBLE PRECISION T, U(2,*), UDOT(2,*), RPAR(*)
245      INTEGER*4 IPAR(*), IER
246      C
247      INTEGER ILEFT, IRIGHT
248      INTEGER*4 JX, JY, MX, MY, MM, IBLOKO, IBLOK, IDN, IUP
249      DOUBLE PRECISION Q1, Q2, Q3, Q4, A3, A4, OM, C3, DY, HDCO
250      DOUBLE PRECISION VDCO, HACO
251      DOUBLE PRECISION C1, C2, C1DN, C2DN, C1UP, C2UP, C1LT, C2LT
252      DOUBLE PRECISION C1RT, C2RT, CYDN, CYUP, HORD1, HORD2, HORAD1
253      DOUBLE PRECISION HORAD2, QQ1, QQ2, QQ3, QQ4, RKIN1, RKIN2, S
254      DOUBLE PRECISION VERTD1, VERTD2, YDN, YUP
255      C
256      C Extract constants from IPAR and RPAR
257      MX = IPAR(1)
258      MY = IPAR(2)
259      MM = IPAR(3)
260      C
261      Q1 = RPAR(1)
262      Q2 = RPAR(2)
263      Q3 = RPAR(3)
264      Q4 = RPAR(4)
265      A3 = RPAR(5)
266      A4 = RPAR(6)
267      OM = RPAR(7)
268      C3 = RPAR(8)
269      DY = RPAR(9)
270      HDCO = RPAR(10)
271      VDCO = RPAR(11)
272      HACO = RPAR(12)
273      C
274      C Set diurnal rate coefficients.
275      S = SIN(OM * T)
276      IF (S .GT. 0.0D0) THEN
277          Q3 = EXP(-A3 / S)
278          Q4 = EXP(-A4 / S)
279      ELSE
280          Q3 = 0.0D0
281          Q4 = 0.0D0
282      ENDIF
283      RPAR(3) = Q3
284      RPAR(4) = Q4
285      C
286      C Loop over all grid points.
287      DO JY = 1, MY
288          YDN = 30.0D0 + (JY - 1.5D0) * DY
289          YUP = YDN + DY
290          CYDN = VDCO * EXP(0.2D0 * YDN)
291          CYUP = VDCO * EXP(0.2D0 * YUP)
292          IBLOKO = (JY - 1) * MX
293          IDN = -MX

```

```

294         IF (JY .EQ. 1) IDN = MX
295         IUP = MX
296         IF (JY .EQ. MY) IUP = -MX
297         DO JX = 1, MX
298             IBLOK = IBLOK0 + JX
299             C1 = U(1,IBLOK)
300             C2 = U(2,IBLOK)
301     C      Set kinetic rate terms.
302             QQ1 = Q1 * C1 * C3
303             QQ2 = Q2 * C1 * C2
304             QQ3 = Q3 * C3
305             QQ4 = Q4 * C2
306             RKIN1 = -QQ1 - QQ2 + 2.0D0 * QQ3 + QQ4
307             RKIN2 = QQ1 - QQ2 - QQ4
308     C      Set vertical diffusion terms.
309             C1DN = U(1,IBLOK + IDN)
310             C2DN = U(2,IBLOK + IDN)
311             C1UP = U(1,IBLOK + IUP)
312             C2UP = U(2,IBLOK + IUP)
313             VERTD1 = CYUP * (C1UP - C1) - CYDN * (C1 - C1DN)
314             VERTD2 = CYUP * (C2UP - C2) - CYDN * (C2 - C2DN)
315     C      Set horizontal diffusion and advection terms.
316             ILEFT = -1
317             IF (JX .EQ. 1) ILEFT = 1
318             IRIGHT = 1
319             IF (JX .EQ. MX) IRIGHT = -1
320             C1LT = U(1,IBLOK + ILEFT)
321             C2LT = U(2,IBLOK + ILEFT)
322             C1RT = U(1,IBLOK + IRIGHT)
323             C2RT = U(2,IBLOK + IRIGHT)
324             HORD1 = HDCO * (C1RT - 2.0D0 * C1 + C1LT)
325             HORD2 = HDCO * (C2RT - 2.0D0 * C2 + C2LT)
326             HORAD1 = HACO * (C1RT - C1LT)
327             HORAD2 = HACO * (C2RT - C2LT)
328     C      Load all terms into UDOT.
329             UDOT(1,IBLOK) = VERTD1 + HORD1 + HORAD1 + RKIN1
330             UDOT(2,IBLOK) = VERTD2 + HORD2 + HORAD2 + RKIN2
331         ENDDO
332     ENDDO
333     C
334     IER = 0
335     C
336     RETURN
337     END
338
339     C      -----
340
341     SUBROUTINE FCVPSET(T, U, FU, JOK, JCUR, GAMMA, H,
342 &                    IPAR, RPAR, V1, V2, V3, IER)
343     C      Routine to set and preprocess block-diagonal preconditioner.
344     C      Note: The dimensions in /BDJ/ below assume at most 100 mesh points.
345     C
346     IMPLICIT NONE
347     C
348     INTEGER IER, JOK, JCUR
349     DOUBLE PRECISION T, U(2,*), FU(*), GAMMA, H
350     INTEGER*4 IPAR(*)
351     DOUBLE PRECISION RPAR(*), V1(*), V2(*), V3(*)
352     C

```



```

353      INTEGER*4 MX, MY, MM, P_IPP, P_BD, P_P
354      DOUBLE PRECISION Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO
355  C
356      IER = 0
357  C
358  C      Extract constants from IPAR and RPAR
359      MX = IPAR(1)
360      MY = IPAR(2)
361      MM = IPAR(3)
362  C
363      Q1 = RPAR(1)
364      Q2 = RPAR(2)
365      Q3 = RPAR(3)
366      Q4 = RPAR(4)
367      C3 = RPAR(8)
368      DY = RPAR(9)
369      HDCO = RPAR(10)
370      VDCO = RPAR(11)
371  C
372  C      Extract pointers into IPAR and RPAR
373      P_IPP = IPAR(4)
374      P_BD = IPAR(5)
375      P_P = IPAR(6)
376  C
377  C      If needed, recompute BD
378  C
379      IF (JOK .EQ. 1) THEN
380  C      JOK = 1. Use saved BD
381          JCUR = 0
382      ELSE
383  C      JOK = 0. Compute diagonal Jacobian blocks.
384  C      (using q4 value computed on last FCVFUN call).
385          CALL PREC_JAC(MX, MY, MM, U, RPAR(P_BD),
386      &      Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO)
387          JCUR = 1
388      ENDIF
389  C
390  C      Copy BD to P
391      CALL DCOPY(4*MM, RPAR(P_BD), 1, RPAR(P_P), 1)
392  C
393  C      Scale P by -GAMMA
394      CALL DSCAL(4*MM, -GAMMA, RPAR(P_P), 1)
395  C
396  C      Perform LU decomposition
397      CALL PREC_LU(MM, RPAR(P_P), IPAR(P_IPP), IER)
398  C
399      RETURN
400      END
401
402  C      -----
403
404      SUBROUTINE FCVPSOL(T, U, FU, R, Z, GAMMA, DELTA, LR,
405      &      IPAR, RPAR, VTEMP, IER)
406  C      Routine to solve preconditioner linear system.
407  C
408      IMPLICIT NONE
409  C
410      INTEGER IER, LR
411      INTEGER*4 IPAR(*)

```

```

412      DOUBLE PRECISION T, U(*), FU(*), R(*), Z(2,*)
413      DOUBLE PRECISION GAMMA, DELTA, RPAR(*)
414      DOUBLE PRECISION VTEMP(*)
415  C
416      INTEGER*4 MM, P_IPP, P_P
417  C
418      IER = 0
419  C
420  C      Extract constants from IPAR and RPAR
421      MM = IPAR(3)
422  C
423  C      Extract pointers into IPAR and RPAR
424      P_IPP = IPAR(4)
425      P_P = IPAR(6)
426  C
427  C      Copy RHS into Z
428      CALL DCOPY(2*MM, R, 1, Z, 1)
429  C
430  C      Solve the block-diagonal system Px = r using LU factors stored in P
431  C      and pivot data in IPP, and return the solution in Z.
432      CALL PREC_SOL(MM, RPAR(P_P), IPAR(P_IPP), Z)
433
434      RETURN
435      END
436
437  C      -----
438
439      SUBROUTINE PREC_JAC(MX, MY, MM, U, BD,
440 &      Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO)
441  C      Routine to compute diagonal Jacobian blocks
442  C
443      IMPLICIT NONE
444  C
445      INTEGER*4 MX, MY, MM
446      DOUBLE PRECISION U(2,*), BD(2,2,MM)
447      DOUBLE PRECISION Q1, Q2, Q3, Q4, C3, DY, HDCO, VDCO
448  C
449      INTEGER*4 JY, JX, IBLOK, IBLOKO
450      DOUBLE PRECISION C1, C2, CYDN, CYUP, DIAG, YDN, YUP
451  C
452      DO JY = 1, MY
453          YDN = 30.0D0 + (JY - 1.5D0) * DY
454          YUP = YDN + DY
455          CYDN = VDCO * EXP(0.2D0 * YDN)
456          CYUP = VDCO * EXP(0.2D0 * YUP)
457          DIAG = -(CYDN + CYUP + 2.0D0 * HDCO)
458          IBLOKO = (JY - 1) * MX
459          DO JX = 1, MX
460              IBLOK = IBLOKO + JX
461              C1 = U(1,IBLOK)
462              C2 = U(2,IBLOK)
463              BD(1,1,IBLOK) = (-Q1 * C3 - Q2 * C2) + DIAG
464              BD(1,2,IBLOK) = -Q2 * C1 + Q4
465              BD(2,1,IBLOK) = Q1 * C3 - Q2 * C2
466              BD(2,2,IBLOK) = (-Q2 * C1 - Q4) + DIAG
467          ENDDO
468      ENDDO
469
470      RETURN

```

```

471      END
472
473 C      -----
474
475      SUBROUTINE PREC_LU(MM, P, IPP, IER)
476 C      Routine to perform LU decomposition on (P+I)
477 C
478      IMPLICIT NONE
479 C
480      INTEGER IER
481      INTEGER*4 MM, IPP(2,MM)
482      DOUBLE PRECISION P(2,2,MM)
483 C
484      INTEGER*4 I
485 C
486 C      Add identity matrix and do LU decompositions on blocks, in place.
487 DO I = 1, MM
488     P(1,1,I) = P(1,1,I) + 1.0D0
489     P(2,2,I) = P(2,2,I) + 1.0D0
490     CALL DGEFA(P(1,1,I), 2, 2, IPP(1,I), IER)
491     IF (IER .NE. 0) RETURN
492 ENDDO
493 C
494      RETURN
495      END
496
497 C      -----
498
499      SUBROUTINE PREC_SOL(MM, P, IPP, Z)
500 C      Routine for backsolve
501 C
502      IMPLICIT NONE
503 C
504      INTEGER*4 MM, IPP(2,MM)
505      DOUBLE PRECISION P(2,2,MM), Z(2,MM)
506 C
507      INTEGER*4 I
508 C
509 DO I = 1, MM
510     CALL DGESL(P(1,1,I), 2, 2, IPP(1,I), Z(1,I), 0)
511 ENDDO
512
513      RETURN
514      END
515
516 C      -----
517
518      subroutine dgefa(a, lda, n, ipvt, info)
519 c
520      implicit none
521 c
522      integer info, idamax, j, k, kp1, l, nm1, n
523      integer*4 lda, ipvt(1)
524      double precision a(lda,1), t
525 c
526 c      dgefa factors a double precision matrix by gaussian elimination.
527 c
528 c      dgefa is usually called by dgeco, but it can be called
529 c      directly with a saving in time if rcond is not needed.

```

```

530 c      (time for dgeco) = (1 + 9/n)*(time for dgefa) .
531 c
532 c      on entry
533 c
534 c          a          double precision(lda, n)
535 c                      the matrix to be factored.
536 c
537 c          lda        integer
538 c                      the leading dimension of the array  a  .
539 c
540 c          n          integer
541 c                      the order of the matrix  a  .
542 c
543 c      on return
544 c
545 c          a          an upper triangular matrix and the multipliers
546 c                      which were used to obtain it.
547 c                      the factorization can be written  a = l*u  where
548 c                      l  is a product of permutation and unit lower
549 c                      triangular matrices and  u  is upper triangular.
550 c
551 c          ipvt        integer(n)
552 c                      an integer vector of pivot indices.
553 c
554 c          info        integer
555 c                      = 0  normal value.
556 c                      = k  if u(k,k) .eq. 0.0  .  this is not an error
557 c                          condition for this subroutine, but it does
558 c                          indicate that dgesl or dgedi will divide by zero
559 c                          if called.  use rcond  in dgeco for a reliable
560 c                          indication of singularity.
561 c
562 c      linpack. this version dated 08/14/78 .
563 c      cleve moler, university of new mexico, argonne national lab.
564 c
565 c      subroutines and functions
566 c
567 c      blas daxpy,dscal,idamax
568 c
569 c      internal variables
570 c
571 c      gaussian elimination with partial pivoting
572 c
573 c      info = 0
574 c      nm1 = n - 1
575 c      if (nm1 .lt. 1) go to 70
576 c      do 60 k = 1, nm1
577 c          kp1 = k + 1
578 c
579 c          find l = pivot index
580 c
581 c          l = idamax(n - k + 1, a(k,k), 1) + k - 1
582 c          ipvt(k) = l
583 c
584 c          zero pivot implies this column already triangularized
585 c
586 c          if (a(l,k) .eq. 0.0d0) go to 40
587 c
588 c          interchange if necessary

```

```

589 c
590         if (l .eq. k) go to 10
591         t = a(l,k)
592         a(l,k) = a(k,k)
593         a(k,k) = t
594     10     continue
595 c
596 c     compute multipliers
597 c
598         t = -1.0d0 / a(k,k)
599         call dscal(n - k, t, a(k + 1,k), 1)
600 c
601 c     row elimination with column indexing
602 c
603         do 30 j = kp1, n
604             t = a(l,j)
605             if (l .eq. k) go to 20
606             a(l,j) = a(k,j)
607             a(k,j) = t
608     20     continue
609             call daxpy(n - k, t, a(k + 1,k), 1, a(k + 1,j), 1)
610     30     continue
611         go to 50
612     40     continue
613         info = k
614     50     continue
615     60     continue
616     70     continue
617         ipvt(n) = n
618         if (a(n,n) .eq. 0.0d0) info = n
619         return
620     end
621
622 C -----
623
624     subroutine dgesl(a, lda, n, ipvt, b, job)
625 c
626     implicit none
627 c
628     integer lda, n, job, k, kb, l, nm1
629     integer*4 ipvt(1)
630     double precision a(lda,1), b(1), ddot, t
631 c
632 c     dgesl solves the double precision system
633 c     a * x = b or trans(a) * x = b
634 c     using the factors computed by dgeco or dgefa.
635 c
636 c     on entry
637 c
638 c         a           double precision(lda, n)
639 c                     the output from dgeco or dgefa.
640 c
641 c         lda         integer
642 c                     the leading dimension of the array a .
643 c
644 c         n           integer
645 c                     the order of the matrix a .
646 c
647 c         ipvt         integer(n)

```

```

648 c          the pivot vector from dgeco or dgefa.
649 c
650 c      b      double precision(n)
651 c          the right hand side vector.
652 c
653 c      job      integer
654 c          = 0          to solve  a*x = b ,
655 c          = nonzero    to solve  trans(a)*x = b  where
656 c                      trans(a)  is the transpose.
657 c
658 c  on return
659 c
660 c      b      the solution vector  x .
661 c
662 c  error condition
663 c
664 c      a division by zero will occur if the input factor contains a
665 c      zero on the diagonal.  technically this indicates singularity
666 c      but it is often caused by improper arguments or improper
667 c      setting of lda .  it will not occur if the subroutines are
668 c      called correctly and if dgeco has set rcond .gt. 0.0
669 c      or dgefa has set info .eq. 0 .
670 c
671 c  to compute  inverse(a) * c  where  c  is a matrix
672 c  with  p  columns
673 c      call dgeco(a,lda,n,ipvt,rcond,z)
674 c      if (rcond is too small) go to ...
675 c      do 10 j = 1, p
676 c          call dgesl(a,lda,n,ipvt,c(1,j),0)
677 c      10 continue
678 c
679 c  linpack. this version dated 08/14/78 .
680 c  cleve moler, university of new mexico, argonne national lab.
681 c
682 c  subroutines and functions
683 c
684 c  blas daxpy,ddot
685 c
686 c  internal variables
687 c
688 c  nm1 = n - 1
689 c  if (job .ne. 0) go to 50
690 c
691 c      job = 0 , solve  a * x = b
692 c      first solve  l*y = b
693 c
694 c      if (nm1 .lt. 1) go to 30
695 c      do 20 k = 1, nm1
696 c          l = ipvt(k)
697 c          t = b(l)
698 c          if (l .eq. k) go to 10
699 c          b(l) = b(k)
700 c          b(k) = t
701 c      10      continue
702 c          call daxpy(n - k, t, a(k + 1,k), 1, b(k + 1), 1)
703 c      20      continue
704 c      30      continue
705 c
706 c      now solve  u*x = y

```

```

707 c
708     do 40 kb = 1, n
709         k = n + 1 - kb
710         b(k) = b(k) / a(k,k)
711         t = -b(k)
712         call daxpy(k - 1, t, a(1,k), 1, b(1), 1)
713 40     continue
714     go to 100
715 50 continue
716 c
717 c     job = nonzero, solve trans(a) * x = b
718 c     first solve trans(u)*y = b
719 c
720     do 60 k = 1, n
721         t = ddot(k - 1, a(1,k), 1, b(1), 1)
722         b(k) = (b(k) - t) / a(k,k)
723 60     continue
724 c
725 c     now solve trans(l)*x = y
726 c
727     if (nm1 .lt. 1) go to 90
728     do 80 kb = 1, nm1
729         k = n - kb
730         b(k) = b(k) + ddot(n - k, a(k + 1,k), 1, b(k + 1), 1)
731         l = ipvt(k)
732         if (l .eq. k) go to 70
733         t = b(l)
734         b(l) = b(k)
735         b(k) = t
736 70     continue
737 80     continue
738 90     continue
739 100 continue
740     return
741     end
742
743 C -----
744
745     subroutine daxpy(n, da, dx, incx, dy, incy)
746 c
747 c     constant times a vector plus a vector.
748 c     uses unrolled loops for increments equal to one.
749 c     jack dongarra, linpack, 3/11/78.
750 c
751     implicit none
752 c
753     integer i, incx, incy, ix, iy, m, mp1
754     integer*4 n
755     double precision dx(1), dy(1), da
756 c
757     if (n .le. 0) return
758     if (da .eq. 0.0d0) return
759     if (incx .eq. 1 .and. incy .eq. 1) go to 20
760 c
761 c     code for unequal increments or equal increments
762 c     not equal to 1
763 c
764     ix = 1
765     iy = 1

```

```

766         if (incx .lt. 0) ix = (-n + 1) * incx + 1
767         if (incy .lt. 0) iy = (-n + 1) * incy + 1
768         do 10 i = 1, n
769             dy(iy) = dy(iy) + da * dx(ix)
770             ix = ix + incx
771             iy = iy + incy
772     10 continue
773     return
774 c
775 c         code for both increments equal to 1
776 c
777 c
778 c         clean-up loop
779 c
780     20 m = mod(n, 4)
781         if ( m .eq. 0 ) go to 40
782         do 30 i = 1, m
783             dy(i) = dy(i) + da * dx(i)
784     30 continue
785         if ( n .lt. 4 ) return
786     40 mp1 = m + 1
787         do 50 i = mp1, n, 4
788             dy(i) = dy(i) + da * dx(i)
789             dy(i + 1) = dy(i + 1) + da * dx(i + 1)
790             dy(i + 2) = dy(i + 2) + da * dx(i + 2)
791             dy(i + 3) = dy(i + 3) + da * dx(i + 3)
792     50 continue
793     return
794 end
795 C -----
796
797     subroutine dscal(n, da, dx, incx)
798 c
799 c         scales a vector by a constant.
800 c         uses unrolled loops for increment equal to one.
801 c         jack dongarra, linpack, 3/11/78.
802 c
803     implicit none
804 c
805     integer i, incx, m, mp1, nincx
806     integer*4 n
807     double precision da, dx(1)
808 c
809     if (n.le.0) return
810     if (incx .eq. 1) go to 20
811 c
812 c         code for increment not equal to 1
813 c
814     nincx = n * incx
815     do 10 i = 1, nincx, incx
816         dx(i) = da * dx(i)
817     10 continue
818     return
819 c
820 c         code for increment equal to 1
821 c
822 c
823 c         clean-up loop
824 c

```



```

825      20 m = mod(n, 5)
826         if ( m .eq. 0 ) go to 40
827         do 30 i = 1, m
828             dx(i) = da * dx(i)
829     30 continue
830         if ( n .lt. 5 ) return
831     40 mp1 = m + 1
832         do 50 i = mp1, n, 5
833             dx(i) = da * dx(i)
834             dx(i + 1) = da * dx(i + 1)
835             dx(i + 2) = da * dx(i + 2)
836             dx(i + 3) = da * dx(i + 3)
837             dx(i + 4) = da * dx(i + 4)
838     50 continue
839         return
840     end
841
842 C      -----
843
844     double precision function ddot(n, dx, incx, dy, incy)
845 C
846 C      forms the dot product of two vectors.
847 C      uses unrolled loops for increments equal to one.
848 C      jack dongarra, linpack, 3/11/78.
849 C
850     implicit none
851 C
852     integer i, incx, incy, ix, iy, m, mp1
853     integer*4 n
854     double precision dx(1), dy(1), dtemp
855 C
856     ddot = 0.0d0
857     dtemp = 0.0d0
858     if (n .le. 0) return
859     if (incx .eq. 1 .and. incy .eq. 1) go to 20
860 C
861 C      code for unequal increments or equal increments
862 C      not equal to 1
863 C
864     ix = 1
865     iy = 1
866     if (incx .lt. 0) ix = (-n + 1) * incx + 1
867     if (incy .lt. 0) iy = (-n + 1) * incy + 1
868     do 10 i = 1, n
869         dtemp = dtemp + dx(ix) * dy(iy)
870         ix = ix + incx
871         iy = iy + incy
872     10 continue
873     ddot = dtemp
874     return
875 C
876 C      code for both increments equal to 1
877 C
878 C
879 C      clean-up loop
880 C
881     20 m = mod(n, 5)
882         if ( m .eq. 0 ) go to 40
883         do 30 i = 1,m

```

```

884         dtemp = dtemp + dx(i) * dy(i)
885 30 continue
886     if ( n .lt. 5 ) go to 60
887 40 mp1 = m + 1
888     do 50 i = mp1, n, 5
889         dtemp = dtemp + dx(i) * dy(i) + dx(i + 1) * dy(i + 1) +
890 *           dx(i + 2) * dy(i + 2) + dx(i + 3) * dy(i + 3) +
891 *           dx(i + 4) * dy(i + 4)
892 50 continue
893 60 ddot = dtemp
894     return
895 end
896
897 C -----
898
899 integer function idamax(n, dx, incx)
900 C
901 C finds the index of element having max. absolute value.
902 C jack dongarra, linpack, 3/11/78.
903 C
904 C implicit none
905 C
906 C integer i, incx, ix
907 C integer*4 n
908 C double precision dx(1), dmax
909 C
910 idamax = 0
911 if (n .lt. 1) return
912 idamax = 1
913 if (n .eq. 1) return
914 if (incx .eq. 1) go to 20
915 C
916 C code for increment not equal to 1
917 C
918 ix = 1
919 dmax = abs(dx(1))
920 ix = ix + incx
921 do 10 i = 2, n
922     if (abs(dx(ix)) .le. dmax) go to 5
923     idamax = i
924     dmax = abs(dx(ix))
925 5 ix = ix + incx
926 10 continue
927 return
928 C
929 C code for increment equal to 1
930 C
931 20 dmax = abs(dx(1))
932 do 30 i = 2, n
933     if (abs(dx(i)) .le. dmax) go to 30
934     idamax = i
935     dmax = abs(dx(i))
936 30 continue
937 return
938 end
939
940 C -----
941
942 subroutine dcopy(n, dx, incx, dy, incy)

```

```

943 c
944 c     copies a vector, x, to a vector, y.
945 c     uses unrolled loops for increments equal to one.
946 c     jack dongarra, linpack, 3/11/78.
947 c
948 c     implicit none
949 c
950 c     integer i, incx, incy, ix, iy, m, mp1
951 c     integer*4 n
952 c     double precision dx(1), dy(1)
953 c
954 c     if (n .le. 0) return
955 c     if (incx .eq. 1 .and. incy .eq. 1) go to 20
956 c
957 c         code for unequal increments or equal increments
958 c         not equal to 1
959 c
960 c         ix = 1
961 c         iy = 1
962 c         if (incx .lt. 0) ix = (-n + 1) * incx + 1
963 c         if (incy .lt. 0) iy = (-n + 1) * incy + 1
964 c         do 10 i = 1, n
965 c             dy(iy) = dx(ix)
966 c             ix = ix + incx
967 c             iy = iy + incy
968 c     10 continue
969 c     return
970 c
971 c         code for both increments equal to 1
972 c
973 c
974 c         clean-up loop
975 c
976 c     20 m = mod(n, 7)
977 c         if ( m .eq. 0 ) go to 40
978 c         do 30 i = 1, m
979 c             dy(i) = dx(i)
980 c     30 continue
981 c         if ( n .lt. 7 ) return
982 c     40 mp1 = m + 1
983 c         do 50 i = mp1, n, 7
984 c             dy(i) = dx(i)
985 c             dy(i + 1) = dx(i + 1)
986 c             dy(i + 2) = dx(i + 2)
987 c             dy(i + 3) = dx(i + 3)
988 c             dy(i + 4) = dx(i + 4)
989 c             dy(i + 5) = dx(i + 5)
990 c             dy(i + 6) = dx(i + 6)
991 c     50 continue
992 c     return
993 c     end

```

H Listing of fcvkryx_bbd_p.f

```

1 C -----
2 C $Revision: 1.3 $
3 C $Date: 2006/03/09 00:10:08 $
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 IMPLICIT NONE
13 C
14 INCLUDE "mpif.h"
15 C
16 INTEGER*4 NLOCAL
17 PARAMETER (NLOCAL=10)
18 C
19 INTEGER NOUT, LNST, LNFE, LNSETUP, LNNI, LNCFL, LNETF, LNPE
20 INTEGER LNLI, LNPS, LNCFL, MYPE, IER, NPES, METH, ITMETH
21 INTEGER LLENRW, LLENIW, LLENRWLS, LLENIWLS
22 INTEGER IATOL, ITASK, IPRE, IGS, JOUT
23 INTEGER*4 IOUT(25), IPAR(2)
24 INTEGER*4 NEQ, I, MUDQ, MLDQ, MU, ML, NETF
25 INTEGER*4 NST, NFE, NPSET, NPE, NPS, NNI, NLI, NCFN, NCFL, NGBBBD
26 INTEGER*4 LENRW, LENIW, LENRWLS, LENIWLS, LENRWBBD, LENIWBBD
27 DOUBLE PRECISION Y(1024), ROUT(10), RPAR(1)
28 DOUBLE PRECISION ALPHA, TOUT, ERMAX, AVDIM
29 DOUBLE PRECISION ATOL, ERRI, RTOL, GERMAX, DTOUT, T
30 C
31 DATA ATOL/1.0D-10/, RTOL/1.0D-5/, DTOUT/0.1D0/, NOUT/10/
32 DATA LLENRW/1/, LLENIW/2/, LNST/3/, LNFE/4/, LNETF/5/, LNCFL/6/,
33 1 LNNI/7/, LNSETUP/8/, LLENRWLS/13/, LLENIWLS/14/,
34 1 LENPE/18/, LNLI/20/, LNPS/19/, LNCFL/21/
35 C
36 C Get NPES and MYPE. Requires initialization of MPI.
37 CALL MPI_INIT(IER)
38 IF (IER .NE. 0) THEN
39 WRITE(6,5) IER
40 5 FORMAT(///' MPI_ERROR: MPI_INIT returned IER = ', I5)
41 STOP
42 ENDIF
43 CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IER)
44 IF (IER .NE. 0) THEN
45 WRITE(6,6) IER
46 6 FORMAT(///' MPI_ERROR: MPI_COMM_SIZE returned IER = ', I5)
47 CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
48 STOP
49 ENDIF
50 CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYPE, IER)
51 IF (IER .NE. 0) THEN
52 WRITE(6,7) IER
53 7 FORMAT(///' MPI_ERROR: MPI_COMM_RANK returned IER = ', I5)
54 CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
55 STOP
56 ENDIF
57 C

```

```

58 C      Set input arguments.
59      NEQ = NPES * NLOCAL
60      T = 0.0D0
61      METH = 2
62      ITMETH = 2
63      IATOL = 1
64      ITASK = 1
65      IPRE = 1
66      IGS = 1
67 C      Set parameter alpha
68      ALPHA = 10.0D0
69 C
70 C      Load IPAR and RPAR
71      IPAR(1) = NLOCAL
72      IPAR(2) = MYPE
73      RPAR(1) = ALPHA
74 C
75      DO I = 1, NLOCAL
76          Y(I) = 1.0D0
77      ENDDO
78 C
79      IF (MYPE .EQ. 0) THEN
80          WRITE(6,15) NEQ, ALPHA, RTOL, ATOL, NPES
81 15      FORMAT('Diagonal test problem: '// NEQ = ', I3, /
82          &      ' parameter alpha = ', F8.3/
83          &      ' ydot_i = -alpha*i * y_i (i = 1,...,NEQ)'/
84          &      ' RTOL, ATOL = ', 2E10.1/
85          &      ' Method is BDF/NEWTON/SPGMR'/
86          &      ' Preconditioner is band-block-diagonal, using CVBBDPRE'
87          &      '/ Number of processors = ', I3/)
88      ENDIF
89 C
90      CALL FNVINITP(MPI_COMM_WORLD, 1, NLOCAL, NEQ, IER)
91 C
92      IF (IER .NE. 0) THEN
93          WRITE(6,20) IER
94 20      FORMAT('SUNDIALS_ERROR: FNVINITP returned IER = ', I5)
95          CALL MPI_FINALIZE(IER)
96          STOP
97      ENDIF
98 C
99      CALL FCVMALLOC(T, Y, METH, ITMETH, IATOL, RTOL, ATOL,
100 &      IOUT, ROUT, IPAR, RPAR, IER)
101 C
102      IF (IER .NE. 0) THEN
103          WRITE(6,30) IER
104 30      FORMAT('SUNDIALS_ERROR: FCVMALLOC returned IER = ', I5)
105          CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
106          STOP
107      ENDIF
108 C
109      MUDQ = 0
110      MLDQ = 0
111      MU = 0
112      ML = 0
113      CALL FCVBBDINIT(NLOCAL, MUDQ, MLDQ, MU, ML, 0.0D0, IER)
114      IF (IER .NE. 0) THEN
115          WRITE(6,35) IER
116 35      FORMAT('SUNDIALS_ERROR: FCVBBDINIT returned IER = ', I5)

```

```

117         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
118         STOP
119     ENDIF
120 C
121     CALL FCVBBDSPGMR(IPRE, IGS, 0, 0.0D0, IER)
122     IF (IER .NE. 0) THEN
123         WRITE(6,36) IER
124 36     FORMAT(///' SUNDIALS_ERROR: FCVBBDSPGMR returned IER = ', I5)
125         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
126         STOP
127     ENDIF
128 C
129     IF (MYPE .EQ. 0) WRITE(6,38)
130 38     FORMAT(/'Preconditioning on left'/)
131 C
132 C     Looping point for cases IPRE = 1 and 2.
133 C
134 40     CONTINUE
135 C
136 C     Loop through tout values, call solver, print output, test for failure.
137     TOUT = DTOUT
138     DO 60 JOUT = 1, NOUT
139 C
140         CALL FCVODE(TOUT, T, Y, ITASK, IER)
141 C
142         IF (MYPE .EQ. 0) WRITE(6,45) T, IOUT(LNST), IOUT(LNFE)
143 45     FORMAT(' t = ', E10.2, 5X, 'no. steps = ', I5,
144 &          ' no. f-s = ', I5)
145 C
146         IF (IER .NE. 0) THEN
147             WRITE(6,50) IER, IOUT(15)
148 50     FORMAT(///' SUNDIALS_ERROR: FCVODE returned IER = ', I5, /,
149 &          ' Linear Solver returned IER = ', I5)
150             CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
151             STOP
152         ENDIF
153 C
154         TOUT = TOUT + DTOUT
155 60     CONTINUE
156 C
157 C     Get max. absolute error in the local vector.
158     ERMAX = 0.0D0
159     DO 65 I = 1, NLOCAL
160         ERRI = Y(I) - EXP(-ALPHA * (MYPE * NLOCAL + I) * T)
161         ERMAX = MAX(ERMAX, ABS(ERRI))
162 65     CONTINUE
163 C     Get global max. error from MPI_REDUCE call.
164     CALL MPI_REDUCE(ERMAX, GERMAX, 1, MPI_DOUBLE_PRECISION, MPI_MAX,
165 &          0, MPI_COMM_WORLD, IER)
166     IF (IER .NE. 0) THEN
167         WRITE(6,70) IER
168 70     FORMAT(///' MPI_ERROR: MPI_REDUCE returned IER = ', I5)
169         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
170         STOP
171     ENDIF
172     IF (MYPE .EQ. 0) WRITE(6,75) GERMAX
173 75     FORMAT(/'Max. absolute error is', E10.2/)
174 C
175 C     Print final statistics.

```

```

176      IF (MYPE .EQ. 0) THEN
177          NST = IOUT(LNST)
178          NFE = IOUT(LNFE)
179          NPSET = IOUT(LNSETUP)
180          NPE = IOUT(LNPE)
181          NPS = IOUT(LNPS)
182          NNI = IOUT(LNNI)
183          NLI = IOUT(LNLI)
184          AVDIM = DBLE(NLI) / DBLE(NNI)
185          NCFN = IOUT(LNCF)
186          NCFL = IOUT(LNCFL)
187          NETF = IOUT(LNETF)
188          LENRW = IOUT(LLENRW)
189          LENIW = IOUT(LLENIW)
190          LENRWLS = IOUT(LLENRWLS)
191          LENIWLS = IOUT(LLENIWLS)
192          WRITE(6,80) NST, NFE, NPSET, NPE, NPS, NNI, NLI, AVDIM, NCFN,
193      &          NCFL, NETF, LENRW, LENIW, LENRWLS, LENIWLS
194 80      FORMAT(/'Final statistics:'//
195      &          ' number of steps           = ', I5, 4X,
196      &          ' number of f evals.         = ', I5/,
197      &          ' number of prec. setups = ', I5/,
198      &          ' number of prec. evals. = ', I5, 4X,
199      &          ' number of prec. solves = ', I5/,
200      &          ' number of nonl. iters. = ', I5, 4X,
201      &          ' number of lin. iters. = ', I5/,
202      &          ' average Krylov subspace dimension (NLI/NNI) = ', F8.4/,
203      &          ' number of conv. failures.. nonlinear = ', I3,
204      &          ' linear = ', I3/,
205      &          ' number of error test failures = ', I3/,
206      &          ' main solver real/int workspace sizes = ', 2I5/,
207      &          ' linear solver real/int workspace sizes = ', 2I5)
208      CALL FCVBBD OPT(LENRWBBD, LENIWBBBD, NGEBBBD)
209      WRITE(6,82) LENRWBBD, LENIWBBBD, NGEBBBD
210 82      FORMAT('In CVBBDPRE:'//
211      &          ' real/int local workspace = ', 2I5/
212      &          ' number of g evals. = ', I5)
213      ENDIF
214  C
215  C      If IPRE = 1, re-initialize T, Y, and the solver, and loop for
216  C      case IPRE = 2. Otherwise jump to final block.
217      IF (IPRE .EQ. 2) GO TO 99
218  C
219      T = 0.0D0
220      DO I = 1, NLOCAL
221          Y(I) = 1.0D0
222      ENDDO
223  C
224      CALL FCVREINIT(T, Y, IATOL, RTOL, ATOL, IER)
225      IF (IER .NE. 0) THEN
226          WRITE(6,91) IER
227 91      FORMAT(///' SUNDIALS_ERROR: FCVREINIT returned IER = ', I5)
228          CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
229          STOP
230      ENDIF
231  C
232      IPRE = 2
233  C
234      CALL FCVBBDREINIT(NLOCAL, MUDQ, MLDQ, 0.0D0, IER)

```

```

235     IF (IER .NE. 0) THEN
236         WRITE(6,92) IER
237 92     FORMAT(///' SUNDIALS_ERROR: FCVBBDREINIT returned IER = ', I5)
238         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
239         STOP
240     ENDIF
241 C
242     CALL FCVSPGMRREINIT(IPRE, IGS, 0.0D0, IER)
243     IF (IER .NE. 0) THEN
244         WRITE(6,93) IER
245 93     FORMAT(///' SUNDIALS_ERROR: FCVSPGMRREINIT returned IER = ', I5)
246         CALL MPI_ABORT(MPI_COMM_WORLD, 1, IER)
247         STOP
248     ENDIF
249 C
250     IF (MYPE .EQ. 0) WRITE(6,95)
251 95     FORMAT(//60(' - '))// 'Preconditioning on right'//
252     GO TO 40
253 C
254 C     Free the memory and finalize MPI.
255 99     CALL FCVBBDFFREE
256         CALL FCVFFREE
257         CALL MPI_FINALIZE(IER)
258 C
259     STOP
260     END
261 C
262 C     -----
263 C
264     SUBROUTINE FCVFUN(T, Y, YDOT, IPAR, RPAR, IER)
265 C     Routine for right-hand side function f
266     IMPLICIT NONE
267 C
268     INTEGER*4 IPAR(*), IER
269     DOUBLE PRECISION T, Y(*), YDOT(*), RPAR(*)
270 C
271     INTEGER MYPE
272     INTEGER*4 I, NLOCAL
273     DOUBLE PRECISION ALPHA
274 C
275     NLOCAL = IPAR(1)
276     MYPE = IPAR(2)
277     ALPHA = RPAR(1)
278 C
279     DO I = 1, NLOCAL
280         YDOT(I) = -ALPHA * (MYPE * NLOCAL + I) * Y(I)
281     ENDDO
282 C
283     IER = 0
284 C
285     RETURN
286     END
287 C
288 C     -----
289 C
290     SUBROUTINE FCVGLOCFN(NLOC, T, YLOC, GLOC, IPAR, RPAR, IER)
291 C     Routine to define local approximate function g, here the same as f.
292     IMPLICIT NONE
293 C

```



```

294      INTEGER*4 NLOC, IPAR(*), IER
295      DOUBLE PRECISION T, YLOC(*), GLOC(*), RPAR(*)
296      C
297      CALL FCVFUN(T, YLOC, GLOC, IPAR, RPAR, IER)
298      C
299      RETURN
300      END
301      C
302      C -----
303      C
304      SUBROUTINE FCVCOMMFN(NLOC, T, YLOC, IPAR, RPAR, IER)
305      C Routine to perform communication required for evaluation of g.
306      IER = 0
307      RETURN
308      END

```

