Example Programs for IDA v2.4.0

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

March 24, 2006

UCRL-SM-208113

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 **3**
2.1 A dense example: idadenx 3
2.2 A banded example: idabanx2 5
2.3 A Krylov example: idakryx 8

3 Parallel example problems **11**
3.1 A user preconditioner example: idakryx1_p 11
3.2 An IDABBDPRE preconditioner example: idakryx2_bbd_p 12

4 Fortran example problems **15**
4.1 A serial example: fidadenx 15
4.2 A parallel example: fidakryx_bbd_p 16

References **19**

A Listing of idadenx.c **20**

B Listing of idabanx2.c **27**

C Listing of idakryx.c **39**

D Listing of idakryx1_p.c **49**

E Listing of idakryx2_bbd_p.c **65**

F Listing of fidadenx.f **84**

G Listing of fidakryx_bbd_p.f **89**

1 Introduction

This report is intended to serve as a companion document to the User Documentation of IDA [2]. It provides details, with listings, on the example programs supplied with the IDA distribution package.

The IDA 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.

The IDA distribution contains, in the `sundials/ida/examples_ser` directory, the following four serial examples (using the NVECTOR_SERIAL module):

- `idadenx` solves the Robertson chemical kinetics problem [3], which consists of two differential equations and one algebraic constraint. It also uses the rootfinding feature of IDA.

  The problem is solved with the IDADENSE linear solver using a user-supplied Jacobian.

- `idabanx1` solves a 2-D heat equation, semidiscretized to a DAE on the unit square.

  This program solves the problem with the IDABAND linear solver and the default difference-quotient Jacobian approximation. For purposes of illustration, `IDACalcIC` is called to compute correct values at the boundary, given incorrect values as input initial guesses. The constraint $u > 0.0$ is imposed for all components.

- `idakryx` solves the same 2-D heat equation problem as `idabanx1`, with the Krylov linear solver IDASPGMR. The preconditioner uses only the diagonal elements of the Jacobian.

- `idabanx2` solves a system of PDEs modelling a food web problem, with predator-prey interaction and diffusion, on the unit square in 2-D.

  The PDEs are discretized in space to a system of DAEs which are solved using the IDABAND linear solver with the default difference-quotient Jacobian approximation.

- `idakrydem_lin` solves the same problem as `idakryx`, with three Krylov linear solvers IDASPGMR, IDASPBCG, and IDASPTFQMR. The preconditioner uses only the diagonal elements of the Jacobian.

In the `sundials/ida/examples_par` directory, the IDA distribution contains the following four parallel examples (using the NVECTOR_PARALLEL module):

- `idakryx1_p` solves the same 2-D heat equation problem as `idakryx`, with IDASPGMR in parallel, and with a user-supplied diagonal preconditioner,

- `idakryx1_bbd_p` solves the same problem as `idakryx1_p`.

  This program solves the problem in parallel, using the Krylov linear solver IDASPGMR and the band-block diagonal preconditioner IDABBDPRE with half-bandwidths equal to 1.

- `idakryx2_p` solves the same food web problem as `idabanx2`, but with IDASPGMR and a user-supplied preconditioner.

  The preconditioner supplied to IDASPGMR is the block-diagonal part of the Jacobian with $n_s \times n_s$ blocks arising from the reaction terms only (n_s is the number of species in the model).

- `idakryx2_bbd_p` solves the same food web problem as `idakryx2_p`.

  This program solves the problem using IDASPGMR in parallel and the IDABBDPRE preconditioner.

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

- `fidadenx` is a serial chemical kinetics example (`BDF`/DENSE) with rootfinding, equivalent to `idadenx`.
- `fidakryx_bbd_p` is a parallel example (`BDF`/SPGMR/IDABBDPRE) equivalent to the example `idakryx1_bbd_p`.

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. Solution values may differ within 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%.

In the descriptions below, we make frequent references to the IDA User Document [2]. 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 IDA 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 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 FIDA are automatically pre-processed to generate source code that corresponds to the manner in which the IDA libraries were built (see §4 in this document for more details).

2 Serial example problems

2.1 A dense example: idadenx

This example, due to Robertson [3], is a model of a three-species chemical kinetics system written in DAE form. Differential equations are given for species y_1 and y_2 while an algebraic equation determines y_3. The equations for the system concentrations $y_i(t)$ are:

$$\begin{cases} y_1' & = -.04y_1 + 10^4 y_2 y_3 \\ y_2' & = +.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ 0 & = y_1 + y_2 + y_3 - 1\,. \end{cases} \tag{1}$$

The initial values are taken as $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$ This example computes the three concentration components on the interval from $t = 0$ through $t = 4 \cdot 10^{10}$. While integrating the system, the program 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 IDA.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in IDA header files. The `sundials_types.h` file provides the definition of the type `realtype` (see §5.2 in the user guide [2] for details). For now, it suffices to read `realtype` as `double`. The `ida.h` file provides prototypes for the IDA 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 `IDASolve`. The `ida_dense.h` file provides the prototype for the `IDADense` function. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. Finally, note that `ida_dense.h` also includes the `sundials_dense.h` file which provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements.

This program includes the user-defined accessor macro `IJth` that is useful in writing the problem functions in a form closely matching the mathematical description of the DAE system, i.e. with components numbered from 1 instead of from 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 `DENSE_ELEM` is fully described in §5.6.4.

The program prologue ends with prototypes of the two user-supplied functions that are called by IDA and the prototype of the private function `check_flag` which is used to test the return flag from the IDA user-callable functions.

After various declarations, the `main` program begins by allocating memory for the `yy`, `yp`, and `avtol` vectors using `N_VNew_Serial` with a length argument of `NEQ` (= 3). The lines following that load the initial values of the dependendent variable vectors into `yy` and `yp` and set the realtive tolerance `rtol` and absolute tolerance vector `avtol`. Serial `N_Vector` values are set by first accessing the pointer to their underlying data using the macro `NV_DATA_S` defined by NVECTOR_SERIAL in `nvector_serial.h`.

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

The call to IDACreate creates the IDA solver memory block. The return value of this function is a pointer to the 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 IDA functions.

The call to IDAMalloc allocates the solver memory block. Its arguments include the name of the C function resrob defining the residual function $F(t, y, y')$, and the initial values of t, y, and y'. The argument IDA_SV specifies a vector of absolute tolerances, and this is followed by the address of the relative tolerance rtol and the absolute tolerance vector avtol. See §5.5.1 for full details of this call.

The call to IDARootInit specifies that a rootfinding problem is to be solved along with the integration of the DAE system, that the root functions are specified in the function grob, 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 IDADense (see §5.5.3) and IDADenseSetJacFn (see §5.5.6.4) specify the IDADENSE linear solver with an analytic Jacobian supplied by the user-supplied function jacrob.

The actual solution of the DAE initial value problem is accomplished in the loop over values of the output time tout. In each pass of the loop, the program calls IDASolve in the IDA_NORMAL mode, meaning that the integrator is to take steps until it overshoots tout and then interpolate to $t =$tout, putting the computed value of y(tout) and y'(tout) into yy and yp, respectively, with tret = tout. The return value in this case is IDA_SUCCESS. However, if IDASolve finds a root before reaching the next value of tout, it returns IDA_ROOT_RETURN and stores the root location in tret and the solution there in yy and yp. In either case, the program prints t (= tret) and yy, and also the cumulative number of steps taken so far, and the current method order and step size. In the case of a root, the program calls IDAGetRootInfo to get a length-2 array rootsfound of bits showing which root function was found to have a root. If IDASolve returned any negative value (indicating a failure), the program breaks out of the loop. In the case of a IDA_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 §5.5.5 for full details of the call to IDASolve.

Finally, the main program calls PrintFinalStats to extract and print several relevant statistical quantities, such as the total number of steps, the number of residual and Jacobian evaluations, and the number of error test and convergence test failures. It then calls IDAFree to free the IDA memory block and N_VDestroy_Serial to free the vectors yy, yp, and avtol.

The function PrintFinalStats used here is actually suitable for general use in applications of IDA to any problem with a dense Jacobian. It calls various IDAGet*** and IDADenseGet*** functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (nst), the number of residual evaluations (nre) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (nreLS), the number of Jacobian evaluations (nje), the number of nonlinear (Newton) iterations (nni), the number of local error test failures (netf), the number of nonlinear convergence failures (ncfn), and the number of grob (root function) evaluations (nge). These optional outputs are described in §5.5.8.

The functions resrob (of type IDAResFn) and jacrob (of type IDADenseJacFn) are straightforward expressions of the DAE system (1) and its system Jacobian. The function jacrob makes use of the macro IJth discussed above. See §5.6.1 for detailed specifications of IDAResFn. Similarly, the function grob 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 grob function.

The output generated by `idadenx` 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.

idadenx sample output

```
idadenx: Robertson kinetics DAE serial example problem for IDA
         Three equation chemical kinetics problem.

Linear solver: IDADENSE, with user-supplied Jacobian.
Tolerance parameters:  rtol = 0.0001   atol = 1e-08 1e-14 1e-06
Initial conditions y0 = (1 0 0)
Constraints and id not used.

------------------------------------------------------------------------
  t             y1            y2            y3      | nst  k      h
------------------------------------------------------------------------
2.6403e-01   9.8997e-01   3.4706e-05   1.0000e-02 |  85  2   6.4537e-02
    rootsfound[] =   0   1
4.0000e-01   9.8517e-01   3.3864e-05   1.4796e-02 |  88  2   6.4537e-02
4.0000e+00   9.0550e-01   2.2403e-05   9.4473e-02 | 102  4   4.1426e-01
4.0000e+01   7.1582e-01   9.1851e-06   2.8417e-01 | 136  2   1.3422e+00
4.0000e+02   4.5049e-01   3.2226e-06   5.4950e-01 | 190  4   3.3557e+01
4.0000e+03   1.8321e-01   8.9429e-07   8.1679e-01 | 239  4   3.4533e+02
4.0000e+04   3.8984e-02   1.6218e-07   9.6102e-01 | 287  5   2.0140e+03
4.0000e+05   4.9389e-03   1.9852e-08   9.9506e-01 | 339  3   1.6788e+04
4.0000e+06   5.1683e-04   2.0684e-09   9.9948e-01 | 444  4   2.1755e+05
2.0793e+07   1.0000e-04   4.0004e-10   9.9990e-01 | 495  4   1.0146e+06
    rootsfound[] =   1   0
4.0000e+07   5.2036e-05   2.0816e-10   9.9995e-01 | 506  5   2.5503e+06
4.0000e+08   5.2103e-06   2.0841e-11   9.9999e-01 | 541  4   2.3847e+07
4.0000e+09   5.2125e-07   2.0850e-12   1.0000e-00 | 569  4   3.9351e+08
4.0000e+10   5.1091e-08   2.0437e-13   1.0000e-00 | 589  2   6.0246e+09

Final Run Statistics:

Number of steps                    = 589
Number of residual evaluations     = 832
Number of Jacobian evaluations     = 79
Number of nonlinear iterations     = 832
Number of error test failures      = 14
Number of nonlinear conv. failures = 0
Number of root fn. evaluations     = 631
```

2.2 A banded example: idabanx2

This example is a model of a multi-species food web [1], in which predator-prey relationships with diffusion in a 2-D spatial domain are simulated. Here we consider a model with $s = 2p$ species: p predators and p prey. Species $1, \ldots, p$ (the prey) satisfy rate equations, while species $p+1, \ldots, s$ (the predators) have infinitely fast reaction rates. The coupled PDEs for the species concentrations $c^i(x, y, t)$ are:

$$\begin{cases} \partial c^i/\partial t = R_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) & i = 1, 2, \ldots, p \\ \qquad 0 = R_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) & i = p+1, \ldots, s \ , \end{cases} \tag{2}$$

with

$$R_i(x, y, c) = c^i \left(b_i + \sum_{j=1}^{s} a_{ij} c^j \right) .$$

Here c denotes the vector $\{c^i\}$. The interaction and diffusion coefficients (a_{ij}, b_i, d_i) can be functions of (x, y) in general. The choices made for this test problem are as follows:

$$a_{ij} = \begin{cases} -1 & i = j \\ -0.5 \cdot 10^{-6} & i \le p,\ j > p \\ 10^4 & i > p,\ j \le p \\ 0 & \text{all other } (i, j) , \end{cases}$$

$$b_i = b_i(x, y) = \begin{cases} (1 + \alpha xy + \beta \sin(4\pi x) \sin(4\pi y)) & i \le p \\ -(1 + \alpha xy + \beta \sin(4\pi x) \sin(4\pi y)) & i > p , \end{cases}$$

and

$$d_i = \begin{cases} 1 & i \le p \\ 0.5 & i > p . \end{cases}$$

The spatial domain is the unit square $0 \le x, y \le 1$, and the time interval is $0 \le t \le 1$. The boundary conditions are of homogeneous Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $\alpha = \beta = 0$ [1]. Empirically, a stable equilibrium appears to exist for (2) when α and β are positive, although it may not be unique. In this problem we take $\alpha = 50$ and $\beta = 1000$. For the initial conditions, we set each prey concentration to a simple polynomial profile satisfying the boundary conditions, while the predator concentrations are all set to a flat value:

$$c^i(x, y, 0) = \begin{cases} 10 + i[16x(1 - x)y(1 - y)]^2 & i \le p , \\ 10^5 & i > p . \end{cases}$$

We discretize this PDE system (2) (plus boundary conditions) with central differencing on an $L \times L$ mesh, so as to obtain a DAE system of size $N = sL^2$. The dependent variable vector C consists of the values $c^i(x_j, y_k, t)$ grouped first by species index i, then by x, and lastly by y. At each spatial mesh point, the system has a block of p ODE's followed by a block of p algebraic equations, all coupled. For this example, we take $p = 1, s = 2$, and $L = 20$. The Jacobian is banded, with half-bandwidths `mu = ml` $= sL = 40$.

The `idabanx2.c` program (listed in Appendix B) includes the file `ida_band.h` in order to use the IDABAND linear solver. This file contains the prototype for the `IDABand` routine, the definition for the band matrix type `BandMat`, and the `BAND_COL` and `BAND_COL_ELEM` macros for accessing matrix elements. See §9.2. The main IDA header file `ida.h` is included for the prototypes of the solver user-callable functions and IDA constants, while the file `nvector_serial.h` is included for the definition of the serial `N_Vector` type. The header file `sundials_smalldense.h` is included for the `denalloc` function used in allocating memory for the user data structure.

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 relative and absolute tolerances `RTOL` and `ATOL`, and various parameters for the food-web problem.

Spatial discretization of the PDE naturally produces a DAE system in which equations are numbered by mesh coordinates (i, j). The user-defined macro `IJth_Vptr` 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. `IJ_Vptr(v,i,j)` returns a pointer to the location in `v` corresponding to the species with index `is` $= 0$, x-index `ix` $= i$, and y-index `jy` $= j$.

The type `UserData` is a pointer to a structure containing problem data used in the `resweb` function. This structure is allocated and initialized at the beginning of `main`. The pointer to it, called `webdata`, is then passed to `IDASetRData` and as a result it will be passed back to the `resweb` function each time it is called.

The `main` program is straightforward and very similar to that for `idadenx`. The differences come from the use of the IDABAND linear solver and from the use of the consistent initial conditions algorithm in IDA to correct the initial values. `IDACalcIC` is called with the option `IDA_YA_YDP_INIT`, meaning that IDA is to compute the algebraic components of y and differential components of y', given the differential components of y. This option requires that the `N_Vector id` be set through a call to `IDASetId` specifying the differential and algebraic components. In this example, `id` has components equal to 1 for the prey (indicating differential variables) and 0 for the predators (algebraic variables).

Next, the `IDASolve` function is called in a loop over the output times, and the solution for the species concentrations at the bottom-left and top-right corners is printed, along with the cumulative number of time steps, current method order, and current step size.

Finally, the main program calls `PrintFinalStats` to get and print all of the relevant statistical quantities. It then calls `N_VDestroy_Serial` to free the vectors `cc`, `cp`, and `id`, and `IDAFree` to free the IDA memory block.

The function `PrintFinalStats` used here is actually suitable for general use in applications of IDA to any problem with a banded Jacobian. It calls various `IDAGet***` and `IDABandGet***` functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (`nst`), the number of residual evaluations (`nre`) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (`nreLS`), the number of Jacobian evaluations (`nje`), the number of nonlinear (Newton) iterations (`nni`), the number of local error test failures (`netf`), and the number of nonlinear convergence failures (`ncfn`). These optional outputs are described in §5.5.8.

The function `resweb` is a direct translation of the residual of (2). It first calls the private function `Fweb` to initialize the residual vector with the right-hand side of (2) and then it loops over all grid points, setting residual values appropriately for differential or algebraic components. The calculation of the interaction terms R_i is done in the function `WebRates`.

Sample output from `idabanx2` follows.

idabanx2 sample output

```
idabanx2: Predator-prey DAE serial example problem for IDA

Number of species ns: 2     Mesh dimensions: 20 x 20     System size: 800
Tolerance parameters:  rtol = 1e-05   atol = 1e-05
Linear solver: IDABAND,  Band parameters mu = 40, ml = 40
CalcIC called to correct initial predator concentrations.

------------------------------------------------------------
  t         bottom-left  top-right    | nst  k      h
```

```
------------------------------------------------------------

0.00e+00    1.0000e+01    9.9949e+04    |    0   0    1.6310e-08
            9.9999e+04    9.9949e+04    |

1.00e-03    1.0318e+01    1.0822e+05    |   32   4    1.0823e-04
            1.0319e+05    1.0822e+05    |

1.00e-02    1.6189e+02    1.9735e+06    |  135   4    1.7964e-04
            1.6189e+06    1.9735e+06    |

1.00e-01    2.4019e+02    2.7072e+06    |  231   1    4.4212e-02
            2.4019e+06    2.7072e+06    |

4.00e-01    2.4019e+02    2.7072e+06    |  233   1    1.7685e-01
            2.4019e+06    2.7072e+06    |

7.00e-01    2.4019e+02    2.7072e+06    |  234   1    3.5370e-01
            2.4019e+06    2.7072e+06    |

1.00e+00    2.4019e+02    2.7072e+06    |  235   1    7.0740e-01
            2.4019e+06    2.7072e+06    |

------------------------------------------------------------
Final run statistics:

Number of steps                    = 235
Number of residual evaluations     = 3319
Number of Jacobian evaluations     = 36
Number of nonlinear iterations     = 401
Number of error test failures      = 5
Number of nonlinear conv. failures = 0
```

2.3 A Krylov example: idakryx

This example solves a discretized 2D heat PDE problem. The DAE system arises from the Dirichlet boundary condition $u = 0$, along with the differential equations arising from the discretization of the interior of the region.

The domain is the unit square $\Omega = \{0 \le x, y \le 1\}$ and the equations solved are:

$$\begin{cases} \partial u/\partial t = u_{xx} + u_{yy} & (x,y) \in \Omega \\ u = 0 & (x,y) \in \partial\Omega\,. \end{cases} \tag{3}$$

The time interval is $0 \le t \le 10.24$, and the initial conditions are $u = 16x(1-x)y(1-y)$.

We discretize the PDE system (3) (plus boundary conditions) with central differencing on a 10×10 mesh, so as to obtain a DAE system of size $N = 100$. The dependent variable vector u consists of the values $u(x_j, y_k, t)$ grouped first by x, and then by y. Each discrete boundary condition becomes an algebraic equation within the DAE system.

The source for this example is listed in Appendix C. In this case, `ida_spgmr.h` is included for the definitions of constants and function prototypes associated with the SPGMR method.

After various initializations (including a vector of constraints with all components set to 1 imposing all solution components to be non-negative), the main program creates and initializes the IDA memory block and then attaches the IDASPGMR linear solver using the default `MODIFIED_GS` Gram-Scmidt orthogonalization algorithm.

The calls to `IDASpgmrSetPrecSetupFn` and `IDASpgmrSetPsolveFn` specify the use of the user-supplied preconditioner with `data` being the pointer to user data passed to `PsolveHeat` and `PsetupHeat` whenever they are called (specified with the call to `IDASpgmrSetPrecData`). In a loop over the desired output times, `IDASolve` is called in `IDA_NORMAL` mode and the maximum solution norm is printed.

The `main` program then re-initializes the IDA solver and the IDASPGMR linear solver and solves the problem again, this time using the `CLASSICAL_GS` Gramm-Schmidt orthogonalization algorithm. Finally, memory for the IDA solver and for the various vectors used is deallocated.

The user-supplied residual function `resHeat`, of type `IDAResFn`, loads the DAE residual with the value of u on the boundary (representing the algebraic equations expressing the boundary conditions of (3)) and with the spatial discretization of the PDE (using central differences) in the rest of the domain.

The user-supplied functions `PsetupHeat` and `PsolveHeat` together define the left preconditoner matrix P approximating the system Jacobian matrix $J = \partial F/\partial u + \alpha \partial F/\partial u'$ (where the DAE system is $F(t, u, u') = 0$), and solve the linear systems $Pz = r$. Preconditioning is done in this case by keeping only the diagonal elements of the J matrix above, storing them as inverses in a vector `pp`, when computed in `PsetupHeat`, for subsequent use in `PsolveHeat`. In this instance, only `cj` $= \alpha$ and `data` (the user data structure) are used from the `PsetupHeat` argument list.

Sample output from `idakryx` follows.

idakryx sample output

```
idakryx: Heat equation, serial example problem for IDA
         Discretized heat equation on 2D unit square.
         Zero boundary conditions, polynomial initial conditions.
         Mesh dimensions: 10 x 10        Total system size: 100

Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
Linear solver: IDASPGMR, preconditioner using diagonal elements.


Case 1: gsytpe = MODIFIED_GS

   Output Summary (umax = max-norm of solution)

  time      umax       k  nst  nni  nje   nre   nreLS    h      npe nps
------------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   0.00000e+00  1   39   51   77    51    77   1.05e-01   12 128
  1.28   0.00000e+00  1   41   53   77    53    77   4.21e-01   14 130
  2.56   0.00000e+00  1   43   55   77    55    77   1.69e+00   16 132
  5.12   0.00000e+00  1   44   56   77    56    77   3.37e+00   17 133
 10.24   0.00000e+00  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
```

```
Linear convergence failures    = 0


Case 2: gstype = CLASSICAL_GS

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nje   nre   nreLS    h      npe nps
------------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
  1.28   5.77661e-20  1   41   53   77    53    77   4.21e-01   14 130
  2.56   1.00105e-19  1   43   55   77    55    77   1.69e+00   16 132
  5.12   3.17879e-19  1   44   56   77    56    77   3.37e+00   17 133
 10.24   9.19490e-19  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

3 Parallel example problems

3.1 A user preconditioner example: idakryx1_p

As an example of using IDA with the parallel MPI NVECTOR_PARALLEL module and the Krylov linear solver IDASPGMR with user-defined preconditioner, we provide the example `idakryx1_p` which solves the same 2-D heat PDE problem as `idakryx`. The source is listed in Appendix D.

In the parallel setting, we can think of the processors as being laid out in a grid of size `NPEX` $\times$ `NPEY`, with each processor computing a subset of the solution vector on a submesh of size `MXSUB` $\times$ `MYSUB`. As a consequence, the computation of the residual vector requires that each processor exchange boundary information (namely the components at all interior subgrid boundaries) with its neighboring processors. The message-passing (implemented in the function `rescomm`) uses blocking sends, non-blocking receives, and receive-waiting, in routines `BSend`, `BRecvPost`, and `BRecvWait`, respectively. The data received from each neighboring processor is then loaded into a work array, `uext`, which contains this ghost cell data along with the local portion of the solution.

The local portion of the residual vector is then computed in the routine `reslocal`, which assumes that all inter-processor communication of data needed to calculate `rr` has already been done. Components at interior subgrid boundaries are assumed to be in the work array `uext`. The local portion of the solution vector `uu` is first copied into `uext`. The diffusion terms are evaluated in terms of the `uext` array, and the residuals are formed. The zero Dirichlet boundary conditions are handled here by including the boundary components in the residual, giving algebraic equations for the discrete boundary conditions.

The preconditioner (`PsolveHeat` and `PsetupHeat`) uses the diagonal elements of the Jacobian only and therefore involves only local calculations.

The `idakryx1_p` main program begins with MPI calls to initialize MPI and to set multiprocessor environment parameters `npes` (number of processes) and `thispe` (local process index). Then the local and global vector lengths are set, the user-data structure `Userdata` is created and initialized, and `N_Vector` variables are created and initialized for the initial conditions (`uu` and `up`), for the vector `id` specifying the differential and algebraic components of the solution vector, and for the preconditioner (`pp`). As in `idakryx`, constraints are passed to IDA through the `N_Vector constraints` and the function `IDASetConstraints`. A temporary `N_Vector res` is also created here, for use only in `SetInitialProfiles`. All components of `constraints` are set to 1.0 indicating that non-negativity constraints are to be imposed on each solution component. In addition, for illustration purposes, `idakryx` also excludes the algebraic components of the solution (specified through the `N_Vector id`) from the error test by calling `IDASetSuppressAlg` with a flag `TRUE`.

Sample output from `idakryx1_p` follows.

idakryx1_p sample output

```
idakryx1_p: Heat equation, parallel example problem for IDA
            Discretized heat equation on 2D unit square.
            Zero boundary conditions, polynomial initial conditions.
            Mesh dimensions: 10 x 10        Total system size: 100

Subgrid dimensions: 5 x 5        Processor array: 2 x 2
Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
```

```
SUPPRESSALG = TRUE to suppress local error testing on all boundary components.
Linear solver: IDASPGMR  Preconditioner: diagonal elements only.

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nli   nre   nreLS    h      npe nps
-----------------------------------------------------------------------
  0.00   9.75461e-01  0    0    0    0     0     0   0.00e+00    0   0
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   0.00000e+00  1   39   51   77    51    77   1.05e-01   12 128
  1.28   0.00000e+00  1   41   53   77    53    77   4.21e-01   14 130
  2.56   0.00000e+00  1   43   55   77    55    77   1.69e+00   16 132
  5.12   0.00000e+00  1   44   56   77    56    77   3.37e+00   17 133
 10.24   0.00000e+00  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

3.2 An IDABBDPRE preconditioner example: idakryx2_bbd_p

In this example, `idakryx2_bbd_p`, we solve the same food web problem as with `idabanx2`, but in parallel and with the IDASPGMR linear solver and using the IDABBDPRE module, which generates and uses a band-block-diagonal preconditioner. The source is listed in Appendix E.

As with `idakryx1_p`, we use a `NPEX` × `NPEY` processor grid, with an `MXSUB` × `MYSUB` submesh on each processor. Again, the residual evaluation begins with the communication of ghost data (in `rescomm`), followed by computation using an extended local array, `cext`, in the `reslocal` routine. The exterior Neumann boundary conditions are explicitly handled here by copying data from the first interior mesh line to the ghost cell locations in `cext`. Then the reaction and diffusion terms are evaluated in terms of the `cext` array, and the residuals are formed.

The Jacobian block on each processor is banded, and the half-bandwidths of that block are both equal to `NUM_SPECIES`·`MXSUB`. This is the value supplied as `mudq` and `mldq` in the call to `IDABBDPrecAlloc`. But in order to reduce storage and computation costs for preconditioning, we supply the values `mukeep` = `mlkeep` = 2 (= `NUM_SPECIES`) 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 function `reslocal` is also passed to the IDABBDPRE preconditioner as the `Gres` argument, while a `NULL` pointer is passed for the `Gcomm` argument (since all required communication for the evaluation of `Gres` was already done for `resweb`).

In the `idakryx2_bbd_p` main program, following MPI initializations and creation of user data block `webdata` and `N_Vector` variables, the initial profiles are set, the IDA memory block is created and allocated, the IDABBDPRE preconditioner is initialized, and the IDASPGMR linear solver is attached to the IDA solver. The call to `IDACalcIC` corrects the initial values so that they are consistent with the DAE algebraic constraints.

In a loop over the desired output times, the main solver function `IDASolve` is called, and selected solution components (at the bottom-left and top-right corners of the computational domain) are collected on processor 0 and printed to `stdout`. The main program ends by printing final solver statistics, freeing memory, and finalizing MPI.

Sample output from `idakryx2_bbd_p` follows.

idakryx2_bbd_p sample output

```
idakryx2_bbd_p: Predator-prey DAE parallel example problem

Number of species ns: 2
Mesh dimensions:      20 x 20
Total system size:    800
Subgrid dimensions:   10 x 10
Processor array:      2 x 2
Tolerance parameters:
 relative tolerance = 1e-05
 absolute tolerance = 1e-05
Linear solver: scaled preconditioned GMRES (IDASPGMR)
 max. Krylov dimension: maxl = 12
Preconditioner: band-block-diagonal (IDABBDPRE)
 mudq = 20,  mldq = 20,  mukeep = 2,  mlkeep = 2
CalcIC called to correct initial predator concentrations

------------------------------------------------------------
  t        bottom-left  top-right    | nst  k      h
------------------------------------------------------------

0.00e+00   1.0000e+01   1.0000e+01   |   0  0   1.6310e-08
           9.9999e+04   9.9949e+04   |

1.00e-03   1.0318e+01   1.0827e+01   |  33  4   9.7404e-05
           1.0319e+05   1.0822e+05   |

1.00e-02   1.6189e+02   1.9735e+02   | 125  3   9.7404e-05
           1.6189e+06   1.9735e+06   |

1.00e-01   2.4019e+02   2.7072e+02   | 201  1   4.0396e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+02   | 204  1   3.2316e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+02   | 205  1   6.4633e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+02   | 205  1   6.4633e-01
           2.4019e+06   2.7072e+06   |

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

Final statistics:

Number of steps                    = 205
Number of residual evaluations     = 1279
Number of nonlinear iterations     = 253
Number of error test failures      = 0
Number of nonlinear conv. failures = 0
```

```
Number of linear iterations         = 1024
Number of linear conv. failures     = 0

Number of preconditioner setups     = 25
Number of preconditioner solves     = 1279
Number of local residual evals.     = 1050
```

4 Fortran example problems

The FORTRAN example problem programs supplied with the IDA 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: fidadenx

The `fidadenx` example is a FORTRAN equivalent of the `idadenx` problem. The source program `fidadenx.f` is listed in Appendix F.

The main program begins with declarations and initializations. It calls the routines `FNVINITS`, `FIDAMALLOC`, `FIDAROOTINIT`, `FIDADENSE`, and `FIDADENSESETJAC`, to initialize the NVECTOR_SERIAL module, the main solver memory, the rootfinding module, and the IDADENSE module, and to specify user-supplied Jacobian routine, respectively. It calls `FIDASOLVE` in a loop over `TOUT` values, with printing of the solution values and performance data (current order and step count from the `IOUT` array, and current step size from the `ROUT` array). In the case of a root return, an extra line is printed with the root information from `FIDAROOTINFO`. At the end, it prints a number of performance counters, and frees memory with calls to `FIDAROOTFREE` and `FIDAFREE`.

In `fidadenx.f`, the `FIDARESFUN` routine is a straightforward implementation of Eqns. (1). In `FIDADJAC`, the 3×3 system Jacobian is supplied. The `FIDAROOTFN` routine defines the two root functions, which are set to determine the points at which $y_1 = 10^{-4}$ or $y_3 = .01$. The final two routines are for the printing of a header and of the final run statistics.

The following is sample output from `fidadenx`. The performance of FIDA here is similar to that of IDA on the `idadenx` problem, with somewhat lower cost counters owing to the larger absolute error tolerances.

fidadenx sample output

```
fidadenx: Robertson kinetics DAE serial exampleproblem for IDA
          Three equation chemicalkinetics problem.

Tolerance parameters:  rtol = 0.10E-03   atol =  0.10E-05 0.10E-09 0.10E-05
Initial conditions y0 = ( 0.10E+01 0.00E+00 0.00E+00)

  t            y1           y2           y3         nst  k    h
0.2640E+00   0.9900E+00   0.3471E-04   0.1000E-01   75  2  0.5716E-01
     Above is a root, INFO() =   0  1
0.4000E+00   0.9852E+00   0.3386E-04   0.1480E-01   77  3  0.1143E+00
0.4000E+01   0.9055E+00   0.2240E-04   0.9447E-01   91  4  0.3704E+00
0.4000E+02   0.7158E+00   0.9185E-05   0.2842E+00  127  4  0.2963E+01
0.4000E+03   0.4505E+00   0.3223E-05   0.5495E+00  177  3  0.1241E+02
0.4000E+04   0.1832E+00   0.8940E-06   0.8168E+00  228  3  0.2765E+03
0.4000E+05   0.3899E-01   0.1622E-06   0.9610E+00  278  5  0.2614E+04
0.4000E+06   0.4939E-02   0.1985E-07   0.9951E+00  324  5  0.2770E+05
0.4000E+07   0.5176E-03   0.2072E-08   0.9995E+00  355  4  0.3979E+06
```

```
0.2075E+08    0.1000E-03    0.4000E-09    0.9999E+00   374   4   0.1592E+07
     Above is a root, INFO() =   1  0
0.4000E+08    0.5191E-04    0.2076E-09    0.9999E+00   380   3   0.6366E+07
0.4000E+09    0.5882E-05    0.2353E-10    0.1000E+01   394   1   0.9167E+08
0.4000E+10    0.7054E-06    0.2822E-11    0.1000E+01   402   1   0.1467E+10
0.4000E+11   -0.7300E-06   -0.2920E-11    0.1000E+01   407   1   0.2347E+11

Final Run Statistics:

Number of steps                    = 407
Number of residual evaluations     = 557
Number of Jacobian evaluations     =  65
Number of nonlinear iterations     = 557
Number of error test failures      =   6
Number of nonlinear conv. failures =   0
Number of root function evals.     = 437
```

4.2 A parallel example: fidakryx_bbd_p

This example, `fidakryx_bbd_p`, is the FORTRAN equivalent of the `idakryx1_bbd_p` example. The heat equation problem is described under the `idakryx` example above, but here it is solved in parallel, using the IDABBDPRE (band-block-diagonal) preconditioner module. The decomposition of the problem onto a processor array is identical to that in the `idakryx1_p` example above. The source file, `fidakryx_bbd_p.f`, is listed in Appendix G.

The problem is solved twice — once with half-bandwidths of 5 in the difference-quotient banded preconditioner blocks, and once with half-bandwidths of 1 (which results in lumping of Jacobian values). In both cases, the retained banded blocks are tridiagonal, even though the true Jacobian is not.

The main program begins with initializations, including MPI calls, a call to `FNVINITP` to initialize NVECTOR_PARALLEL, and a call to `SETINITPROFILE` to initialize the `UU`, `UP`, `ID`, and `CONSTR` arrays (containing the solution vector, solution derivative vector, the differential/algebraic bit vector, and the contraint specification vector, respectively). A call to `FIDASETIIN` and two calls to `FIDASETVIN` are made to suppress error control on the algebraic variables, and to supply the `ID` array and constraints array (making the computed solution non-negative). The call to `FIDAMALLOC` initializes the FIDA main memory, and the calls to `FIDABBDINIT` and `FIDABBDSPGMR` initialize the FIDABBD module.

In the first loop over `TOUT` values, the main program calls `FIDASOLVE` and prints the max-norm of the solution and selected counters. When finished, it calls `PRNTFINALSTATS` to print a few more counters.

The second solution is initialized by a second call to `SETINITPROFILE`, and calls to `FIDAREINIT` and `FIDABBDREINIT`. After completing the second solution, the program frees memory and terminates MPI.

The `FIDARESFUN` routine simply calls two other routines: `FIDACOMMFN`, to communicate boundary needed data from `U` to an extension of it called `UEXT`; and `FIDAGLOCFN`, to compute the residuals in terms of `UEXT` and `UP`.

The following is a sample output from `fidakryx_bbd_p`, with a 10×10 mesh and `NPES` $= 4$ processors. The performance is similar for the two solutions. The second case requires more linear iterations, as expected, but their cost is offset by the much cheaper preconditioner evaluations.

fidakryx_bbd_p sample output

```
fidakryx_bbd_p: Heat equation, parallel example problem for FIDA
                Discretized heat equation on 2D unit square.
                Zero boundary conditions, polynomial conditions.
                Mesh dimensions: 10 x 10         Total system size: 100

Subgrid dimensions:  5 x  5           Processor array:  2 x  2
Tolerance parameters: rtol = 0.00E+00   atol = 0.10E-02
Constraints set to force all solution components >= 0.
SUPPRESSALG = TRUE to remove boundary components from the error test.
Linear solver: SPGMR.    Preconditioner: BBDPRE - Banded-block-diagonal.


Case  1
  Difference quotient half-bandwidths = 5
  Retained matrix half-bandwidths = 1

Output Summary
  umax = max-norm of solution
  nre = nre + nreLS (total number of RES evals.)

   time          umax       k nst   nni   nli    nre    nge       h      npe  nps
-----------------------------------------------------------------------------
 0.1000E-01   0.82411E+00  2  12    14     7   14+ 7    96   0.26E-02    8    21
 0.2000E-01   0.68812E+00  3  15    18    12   18+12    96   0.51E-02    8    30
 0.4000E-01   0.47075E+00  3  18    24    22   24+22   108   0.66E-02    9    46
 0.8000E-01   0.21660E+00  3  22    29    30   29+30   108   0.13E-01    9    59
 0.1600E+00   0.45659E-01  4  28    37    43   37+43   120   0.26E-01   10    80
 0.3200E+00   0.21096E-02  4  35    45    59   45+59   120   0.24E-01   10   104
 0.6400E+00   0.50233E-04  1  40    54    70   54+70   156   0.19E+00   13   124
 0.1280E+01   0.23658E-18  1  42    56    70   56+70   180   0.76E+00   15   126
 0.2560E+01   0.14313E-19  1  43    57    70   57+70   192   0.15E+01   16   127
 0.5120E+01   0.42389E-19  1  44    58    70   58+70   204   0.30E+01   17   128
 0.1024E+02   0.99241E-19  1  45    59    70   59+70   216   0.61E+01   18   129

Error test failures            =  1
Nonlinear convergence failures =  0
Linear convergence failures    =  0


Case  2
  Difference quotient half-bandwidths = 1
  Retained matrix half-bandwidths = 1

Output Summary
  umax = max-norm of solution
  nre = nre + nreLS (total number of RES evals.)

   time          umax       k nst   nni   nli    nre    nge       h      npe  nps
-----------------------------------------------------------------------------
 0.1000E-01   0.82411E+00  2  12    14     7   14+ 7    32   0.26E-02    8    21
 0.2000E-01   0.68812E+00  3  15    18    12   18+12    32   0.51E-02    8    30
 0.4000E-01   0.47093E+00  3  19    23    20   23+20    36   0.10E-01    9    43
 0.8000E-01   0.21655E+00  3  23    27    32   27+32    36   0.10E-01    9    59
 0.1600E+00   0.45225E-01  4  27    33    44   33+44    40   0.20E-01   10    77
 0.3200E+00   0.21868E-02  3  34    41    67   41+67    44   0.41E-01   11   108
 0.6400E+00   0.22218E-19  1  39    49    86   49+86    52   0.16E+00   13   135
 0.1280E+01   0.19350E-19  1  41    51    86   51+86    60   0.66E+00   15   137
```

```
 0.2560E+01   0.16748E-18  1  42   52   86  52+86   64   0.13E+01  16  138
 0.5120E+01   0.13522E-17  1  43   53   86  53+86   68   0.26E+01  17  139
 0.1024E+02   0.10274E-16  1  44   54   86  54+86   72   0.52E+01  18  140

Error test failures            =  0
Nonlinear convergence failures =  0
Linear convergence failures    =  0
```

References

[1] Peter N. Brown. Decay to uniform states in food webs. *SIAM J. Appl. Math.*, 46:376–392, 1986.

[2] A. C. Hindmarsh and R. Serban. User Documentation for IDA v2.4.0. Technical Report UCRL-SM-208112, LLNL, 2005.

[3] H. H. Robertson. The solution of a set of reaction rate equations. In J. Walsh, editor, *Numerical analysis: an introduction*, pages 178–182. Academ. Press, 1966.

A Listing of idadenx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.4 $
 * $Date: 2006/01/24 19:25:00 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * This simple example problem for IDA, due to Robertson,
 * is from chemical kinetics, and consists of the following three
 * equations:
 *
 *      dy1/dt = -.04*y1 + 1.e4*y2*y3
 *      dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*y2**2
 *         0   = y1 + y2 + y3 - 1
 *
 * on the interval from t = 0.0 to t = 4.e10, with initial
 * conditions: y1 = 1, y2 = y3 = 0.
 *
 * While integrating the system, we also use the rootfinding
 * feature to find the points at which y1 = 1e-4 or at which
 * y3 = 0.01.
 *
 * The problem is solved with IDA using IDADENSE for the linear
 * solver, with a user-supplied Jacobian. Output is printed at
 * t = .4, 4, 40, ..., 4e10.
 * -----------------------------------------------------------------
 */

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

#include "ida.h"
#include "nvector_serial.h"
#include "ida_dense.h"
#include "sundials_types.h"
#include "sundials_math.h"

/* Problem Constants */

#define NEQ   3
#define NOUT  12

#define ZERO RCONST(0.0);
#define ONE  RCONST(1.0);

/* Macro to define dense matrix elements, indexed from 1. */

#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1)

/* Prototypes of functions called by IDA */

int resrob(realtype tres, N_Vector yy, N_Vector yp,
           N_Vector resval, void *rdata);

static int grob(realtype t, N_Vector yy, N_Vector yp,
                realtype *gout, void *g_data);
```

```c

int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
           N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
           N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);

/* Prototypes of private functions */
static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y);
static void PrintOutput(void *mem, realtype t, N_Vector y);
static void PrintRootInfo(int root_f1, int root_f2);
static void PrintFinalStats(void *mem);
static int check_flag(void *flagvalue, char *funcname, int opt);

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

int main(void)
{
  void *mem;
  N_Vector yy, yp, avtol;
  realtype rtol, *yval, *ypval, *atval;
  realtype t0, tout1, tout, tret;
  int iout, retval, retvalr;
  int rootsfound[2];

  mem = NULL;
  yy = yp = avtol = NULL;
  yval = ypval = atval = NULL;

  /* Allocate N-vectors. */
  yy = N_VNew_Serial(NEQ);
  if(check_flag((void *)yy, "N_VNew_Serial", 0)) return(1);
  yp = N_VNew_Serial(NEQ);
  if(check_flag((void *)yp, "N_VNew_Serial", 0)) return(1);
  avtol = N_VNew_Serial(NEQ);
  if(check_flag((void *)avtol, "N_VNew_Serial", 0)) return(1);

  /* Create and initialize  y, y', and absolute tolerance vectors. */
  yval  = NV_DATA_S(yy);
  yval[0] = ONE;
  yval[1] = ZERO;
  yval[2] = ZERO;

  ypval = NV_DATA_S(yp);
  ypval[0]  = RCONST(-0.04);
  ypval[1]  = RCONST(0.04);
  ypval[2]  = ZERO;

  rtol = RCONST(1.0e-4);

  atval = NV_DATA_S(avtol);
  atval[0] = RCONST(1.0e-8);
  atval[1] = RCONST(1.0e-14);
  atval[2] = RCONST(1.0e-6);

  /* Integration limits */
  t0 = ZERO;
```

```
117     tout1 = RCONST(0.4);
118
119     PrintHeader(rtol, avtol, yy);
120
121     /* Call IDACreate and IDAMalloc to initialize IDA memory */
122     mem = IDACreate();
123     if(check_flag((void *)mem, "IDACreate", 0)) return(1);
124     retval = IDAMalloc(mem, resrob, t0, yy, yp, IDA_SV, rtol, avtol);
125     if(check_flag(&retval, "IDAMalloc", 1)) return(1);
126
127     /* Free avtol */
128     N_VDestroy_Serial(avtol);
129
130     /* Call IDARootInit to specify the root function grob with 2 components */
131     retval = IDARootInit(mem, 2, grob, NULL);
132     if (check_flag(&retval, "IDARootInit", 1)) return(1);
133
134     /* Call IDADense and set up the linear solver. */
135     retval = IDADense(mem, NEQ);
136     if(check_flag(&retval, "IDADense", 1)) return(1);
137     retval = IDADenseSetJacFn(mem, jacrob, NULL);
138     if(check_flag(&retval, "IDADenseSetJacFn", 1)) return(1);
139
140     /* In loop, call IDASolve, print results, and test for error.
141        Break out of loop when NOUT preset output times have been reached. */
142
143     iout = 0; tout = tout1;
144     while(1) {
145
146       retval = IDASolve(mem, tout, &tret, yy, yp, IDA_NORMAL);
147
148       PrintOutput(mem,tret,yy);
149
150       if(check_flag(&retval, "IDASolve", 1)) return(1);
151
152       if (retval == IDA_ROOT_RETURN) {
153         retvalr = IDAGetRootInfo(mem, rootsfound);
154         check_flag(&retvalr, "IDAGetRootInfo", 1);
155         PrintRootInfo(rootsfound[0],rootsfound[1]);
156       }
157
158       if (retval == IDA_SUCCESS) {
159         iout++;
160         tout *= RCONST(10.0);
161       }
162
163       if (iout == NOUT) break;
164     }
165
166     PrintFinalStats(mem);
167
168     /* Free memory */
169
170     IDAFree(&mem);
171     N_VDestroy_Serial(yy);
172     N_VDestroy_Serial(yp);
173
174     return(0);
175
```

```
176 }
177
178 /*
179  *--------------------------------------------------------------------
180  * Functions called by IDA
181  *--------------------------------------------------------------------
182  */
183
184 /*
185  * Define the system residual function.
186  */
187
188 int resrob(realtype tres, N_Vector yy, N_Vector yp, N_Vector rr, void *rdata)
189 {
190   realtype *yval, *ypval, *rval;
191
192   yval = NV_DATA_S(yy);
193   ypval = NV_DATA_S(yp);
194   rval = NV_DATA_S(rr);
195
196   rval[0]  = RCONST(-0.04)*yval[0] + RCONST(1.0e4)*yval[1]*yval[2];
197   rval[1]  = -rval[0] - RCONST(3.0e7)*yval[1]*yval[1] - ypval[1];
198   rval[0] -=  ypval[0];
199   rval[2]  =  yval[0] + yval[1] + yval[2] - ONE;
200
201   return(0);
202 }
203
204 /*
205  * Root function routine. Compute functions g_i(t,y) for i = 0,1.
206  */
207
208 static int grob(realtype t, N_Vector yy, N_Vector yp, realtype *gout,
209                 void *g_data)
210 {
211   realtype *yval, y1, y3;
212
213   yval = NV_DATA_S(yy);
214   y1 = yval[0]; y3 = yval[2];
215   gout[0] = y1 - RCONST(0.0001);
216   gout[1] = y3 - RCONST(0.01);
217
218   return(0);
219 }
220
221 /*
222  * Define the Jacobian function.
223  */
224
225 int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
226            N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
227            N_Vector tempv1, N_Vector tempv2, N_Vector tempv3)
228 {
229   realtype *yval;
230
231   yval = NV_DATA_S(yy);
232
233   IJth(JJ,1,1) = RCONST(-0.04) - cj;
234   IJth(JJ,2,1) = RCONST(0.04);
```

```
  IJth(JJ,3,1) = ONE;
  IJth(JJ,1,2) = RCONST(1.0e4)*yval[2];
  IJth(JJ,2,2) = RCONST(-1.0e4)*yval[2] - RCONST(6.0e7)*yval[1] - cj;
  IJth(JJ,3,2) = ONE;
  IJth(JJ,1,3) = RCONST(1.0e4)*yval[1];
  IJth(JJ,2,3) = RCONST(-1.0e4)*yval[1];
  IJth(JJ,3,3) = ONE;

  return(0);
}

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

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

static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y)
{
  realtype *atval, *yval;

  atval  = NV_DATA_S(avtol);
  yval  = NV_DATA_S(y);

  printf("\nidadenx: Robertson kinetics DAE serial example problem for IDA\n");
  printf("          Three equation chemical kinetics problem.\n\n");
  printf("Linear solver: IDADENSE, with user-supplied Jacobian.\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg %Lg %Lg \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%Lg %Lg %Lg)\n",
         yval[0], yval[1], yval[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg %lg %lg \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%lg %lg %lg)\n",
         yval[0], yval[1], yval[2]);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g %g %g \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%g %g %g)\n",
         yval[0], yval[1], yval[2]);
#endif
  printf("Constraints and id not used.\n\n");
  printf("-----------------------------------------------------------------------\n");
  printf("  t             y1           y2           y3");
  printf("       | nst  k       h\n");
  printf("-----------------------------------------------------------------------\n");
}

/*
 * Print Output
 */

static void PrintOutput(void *mem, realtype t, N_Vector y)
```

```
{
  realtype *yval;
  int retval, kused;
  long int nst;
  realtype hused;

  yval  = NV_DATA_S(y);

  retval = IDAGetLastOrder(mem, &kused);
  check_flag(&retval, "IDAGetLastOrder", 1);
  retval = IDAGetNumSteps(mem, &nst);
  check_flag(&retval, "IDAGetNumSteps", 1);
  retval = IDAGetLastStep(mem, &hused);
  check_flag(&retval, "IDAGetLastStep", 1);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%10.4Le %12.4Le %12.4Le %12.4Le | %3ld  %1d %12.4Le\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%10.4le %12.4le %12.4le %12.4le | %3ld  %1d %12.4le\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#else
  printf("%10.4e %12.4e %12.4e %12.4e | %3ld  %1d %12.4e\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#endif
}

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

/*
 * Print final integrator statistics
 */

static void PrintFinalStats(void *mem)
{
  int retval;
  long int nst, nni, nje, nre, nreLS, netf, ncfn, nge;

  retval = IDAGetNumSteps(mem, &nst);
  check_flag(&retval, "IDAGetNumSteps", 1);
  retval = IDAGetNumResEvals(mem, &nre);
  check_flag(&retval, "IDAGetNumResEvals", 1);
  retval = IDADenseGetNumJacEvals(mem, &nje);
  check_flag(&retval, "IDADenseGetNumJacEvals", 1);
  retval = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&retval, "IDAGetNumNonlinSolvIters", 1);
  retval = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&retval, "IDAGetNumErrTestFails", 1);
  retval = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&retval, "IDAGetNumNonlinSolvConvFails", 1);
  retval = IDADenseGetNumResEvals(mem, &nreLS);
  check_flag(&retval, "IDADenseGetNumResEvals", 1);
  retval = IDAGetNumGEvals(mem, &nge);
  check_flag(&retval, "IDAGetNumGEvals", 1);

  printf("\nFinal Run Statistics: \n\n");
```

```
353   printf("Number of steps                    = %ld\n", nst);
354   printf("Number of residual evaluations      = %ld\n", nre+nreLS);
355   printf("Number of Jacobian evaluations      = %ld\n", nje);
356   printf("Number of nonlinear iterations      = %ld\n", nni);
357   printf("Number of error test failures       = %ld\n", netf);
358   printf("Number of nonlinear conv. failures  = %ld\n", ncfn);
359   printf("Number of root fn. evaluations      = %ld\n", nge);
360 }
361
362 /*
363  * Check function return value...
364  *   opt == 0 means SUNDIALS function allocates memory so check if
365  *            returned NULL pointer
366  *   opt == 1 means SUNDIALS function returns a flag so check if
367  *            flag >= 0
368  *   opt == 2 means function allocates memory so check if returned
369  *            NULL pointer
370  */
371
372 static int check_flag(void *flagvalue, char *funcname, int opt)
373 {
374   int *errflag;
375   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
376   if (opt == 0 && flagvalue == NULL) {
377     fprintf(stderr,
378             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
379             funcname);
380     return(1);
381   } else if (opt == 1) {
382     /* Check if flag < 0 */
383     errflag = (int *) flagvalue;
384     if (*errflag < 0) {
385       fprintf(stderr,
386               "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
387               funcname, *errflag);
388       return(1);
389     }
390   } else if (opt == 2 && flagvalue == NULL) {
391     /* Check if function returned NULL pointer - no memory allocated */
392     fprintf(stderr,
393             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
394             funcname);
395     return(1);
396   }
397
398   return(0);
399 }
```

B Listing of idabanx2.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.3 $
 * $Date: 2006/03/17 16:58:00 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example program for IDA: Food web problem.
 *
 * This example program (serial version) uses the IDABAND linear
 * solver, and IDACalcIC for initial condition calculation.
 *
 * The mathematical problem solved in this example is a DAE system
 * that arises from a system of partial differential equations after
 * spatial discretization. The PDE system is a food web population
 * model, with predator-prey interaction and diffusion on the unit
 * square in two dimensions. The dependent variable vector is:
 *
 *         1   2         ns
 *   c = (c , c ,  ..., c  ) , ns = 2 * np
 *
 * and the PDE's are as follows:
 *
 *     i              i       i
 *   dc /dt = d(i)*(c     + c   )  +  R (x,y,c)   (i = 1,...,np)
 *                   xx      yy       i
 *
 *              i       i
 *   0 = d(i)*(c     + c   )  +  R (x,y,c)   (i = np+1,...,ns)
 *               xx      yy       i
 *
 *   where the reaction terms R are:
 *
 *                   i             ns         j
 *   R  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                           j=1
 *
 * The number of species is ns = 2 * np, with the first np being
 * prey and the last np being predators. The coefficients a(i,j),
 * b(i), d(i) are:
 *
 *  a(i,i) = -AA   (all i)
 *  a(i,j) = -GG   (i <= np , j >  np)
 *  a(i,j) =  EE   (i >  np, j <= np)
 *  all other a(i,j) = 0
 *  b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i <= np)
 *  b(i) =-BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i  > np)
 *  d(i) = DPREY   (i <= np)
 *  d(i) = DPRED   (i > np)
 *
 * The various scalar parameters required are set using '#define'
 * statements or directly in routine InitUserData. In this program,
 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
 * normal derivative = 0.
 *
 * A polynomial in x and y is used to set the initial values of the
```

```
 * first np variables (the prey variables) at each x,y location,
 * while initial values for the remaining (predator) variables are
 * set to a flat value, which is corrected by IDACalcIC.
 *
 * The PDEs are discretized by central differencing on a MX by MY
 * mesh.
 *
 * The DAE system is solved by IDA using the IDABAND linear solver.
 * Output is printed at t = 0, .001, .01, .1, .4, .7, 1.
 * -----------------------------------------------------------------
 * References:
 * [1] Peter N. Brown and Alan C. Hindmarsh,
 *     Reduced Storage Matrix Methods in Stiff ODE systems, Journal
 *     of Applied Mathematics and Computation, Vol. 31 (May 1989),
 *     pp. 40-91.
 *
 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Using Krylov Methods in the Solution of Large-Scale
 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
 *     (1994), pp. 1467-1488.
 *
 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Consistent Initial Condition Calculation for Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
 *     pp. 1495-1512.
 * -----------------------------------------------------------------
 */

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

#include "ida.h"                 /* Main header file */
#include "nvector_serial.h"      /* Definitions of N_Vector and NV_DATA_S */
#include "ida_band.h"            /* Use IDABAND linear solver */
#include "sundials_smalldense.h" /* Definition of denalloc */
#include "sundials_types.h"      /* Definitions of realtype and booleantype */

/* Problem Constants. */

#define NPREY       1              /* No. of prey (= no. of predators). */
#define NUM_SPECIES 2*NPREY

#define PI          RCONST(3.1415926535898)
#define FOURPI      (RCONST(4.0)*PI)

#define MX          20             /* MX = number of x mesh points      */
#define MY          20             /* MY = number of y mesh points      */
#define NSMX        (NUM_SPECIES * MX)
#define NEQ         (NUM_SPECIES*MX*MY)
#define AA          RCONST(1.0)    /* Coefficient in above eqns. for a  */
#define EE          RCONST(10000.) /* Coefficient in above eqns. for a  */
#define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a  */
#define BB          RCONST(1.0)    /* Coefficient in above eqns. for b  */
#define DPREY       RCONST(1.0)    /* Coefficient in above eqns. for d  */
#define DPRED       RCONST(0.05)   /* Coefficient in above eqns. for d  */
#define ALPHA       RCONST(50.)    /* Coefficient alpha in above eqns.  */
#define BETA        RCONST(1000.)  /* Coefficient beta in above eqns.   */
#define AX          RCONST(1.0)    /* Total range of x variable         */
```

```
#define AY           RCONST(1.0)    /* Total range of y variable         */
#define RTOL         RCONST(1.e-5)  /* Relative tolerance                */
#define ATOL         RCONST(1.e-5)  /* Absolute tolerance                */
#define NOUT         6              /* Number of output times            */
#define TMULT        RCONST(10.0)   /* Multiplier for tout values        */
#define TADD         RCONST(0.3)    /* Increment for tout values         */
#define ZERO         RCONST(0.)
#define ONE          RCONST(1.0)

/*
 * User-defined vector and accessor macro: IJ_Vptr.
 * IJ_Vptr is defined in order to express the underlying 3-D structure of
 * the dependent variable vector from its underlying 1-D storage (an N_Vector).
 * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
 * species index is = 0, x-index ix = i, and y-index jy = j.
 */

#define IJ_Vptr(vv,i,j) (&NV_Ith_S(vv, (i)*NUM_SPECIES + (j)*NSMX))

/* Type: UserData.  Contains problem constants, etc. */

typedef struct {
  long int Neq, ns, np, mx, my;
  realtype dx, dy, **acoef;
  realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES];
  N_Vector rates;
} *UserData;

/* Prototypes for functions called by the IDA Solver. */

static int resweb(realtype time, N_Vector cc, N_Vector cp, N_Vector resval,
                  void *rdata);

/* Prototypes for private Helper Functions. */

static void InitUserData(UserData webdata);
static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               UserData webdata);
static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol);
static void PrintOutput(void *mem, N_Vector c, realtype t);
static void PrintFinalStats(void *mem);
static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate, UserData webdata);
static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata);
static realtype dotprod(long int size, realtype *x1, realtype *x2);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main()
{
  void *mem;
  UserData webdata;
  N_Vector cc, cp, id;
  int iout, retval;
```

```
176     long int mu, ml;
177     realtype rtol, atol, t0, tout, tret;
178
179     mem = NULL;
180     webdata = NULL;
181     cc = cp = id = NULL;
182
183     /* Allocate and initialize user data block webdata. */
184
185     webdata = (UserData) malloc(sizeof *webdata);
186     webdata->rates = N_VNew_Serial(NEQ);
187     webdata->acoef = denalloc(NUM_SPECIES);
188
189     InitUserData(webdata);
190
191     /* Allocate N-vectors and initialize cc, cp, and id. */
192
193     cc  = N_VNew_Serial(NEQ);
194     if(check_flag((void *)cc, "N_VNew_Serial", 0)) return(1);
195
196     cp  = N_VNew_Serial(NEQ);
197     if(check_flag((void *)cp, "N_VNew_Serial", 0)) return(1);
198
199     id  = N_VNew_Serial(NEQ);
200     if(check_flag((void *)id, "N_VNew_Serial", 0)) return(1);
201
202     SetInitialProfiles(cc, cp, id, webdata);
203
204     /* Set remaining inputs to IDAMalloc. */
205
206     t0 = ZERO;
207     rtol = RTOL;
208     atol = ATOL;
209
210     /* Call IDACreate and IDAMalloc to initialize IDA. */
211
212     mem = IDACreate();
213     if(check_flag((void *)mem, "IDACreate", 0)) return(1);
214
215     retval = IDASetRdata(mem, webdata);
216     if(check_flag(&retval, "IDASetRdata", 1)) return(1);
217
218     retval = IDASetId(mem, id);
219     if(check_flag(&retval, "IDASetId", 1)) return(1);
220
221     retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, rtol, &atol);
222     if(check_flag(&retval, "IDAMalloc", 1)) return(1);
223
224     /* Call IDABand to specify the IDA linear solver. */
225
226     mu = ml = NSMX;
227     retval = IDABand(mem, NEQ, mu, ml);
228     if(check_flag(&retval, "IDABand", 1)) return(1);
229
230     /* Call IDACalcIC (with default options) to correct the initial values. */
231
232     tout = RCONST(0.001);
233     retval = IDACalcIC(mem, t0, cc, cp, IDA_YA_YDP_INIT, tout);
234     if(check_flag(&retval, "IDACalcIC", 1)) return(1);
```

```
235
236     /* Print heading, basic parameters, and initial values. */
237
238     PrintHeader(mu, ml, rtol, atol);
239     PrintOutput(mem, cc, ZERO);
240
241     /* Loop over iout, call IDASolve (normal mode), print selected output. */
242
243     for (iout = 1; iout <= NOUT; iout++) {
244
245       retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
246       if(check_flag(&retval, "IDASolve", 1)) return(retval);
247
248       PrintOutput(mem, cc, tret);
249
250       if (iout < 3) tout *= TMULT; else tout += TADD;
251
252     }
253
254     /* Print final statistics and free memory. */
255
256     PrintFinalStats(mem);
257
258     /* Free memory */
259
260     IDAFree(&mem);
261
262     N_VDestroy_Serial(cc);
263     N_VDestroy_Serial(cp);
264     N_VDestroy_Serial(id);
265
266
267     denfree(webdata->acoef);
268     N_VDestroy_Serial(webdata->rates);
269     free(webdata);
270
271     return(0);
272   }
273
274   /* Define lines for readability in later routines */
275
276   #define acoef  (webdata->acoef)
277   #define bcoef  (webdata->bcoef)
278   #define cox    (webdata->cox)
279   #define coy    (webdata->coy)
280
281   /*
282    *--------------------------------------------------------------------
283    * FUNCTIONS CALLED BY IDA
284    *--------------------------------------------------------------------
285    */
286
287   /*
288    * resweb: System residual function for predator-prey system.
289    * This routine calls Fweb to get all the right-hand sides of the
290    * equations, then loads the residual vector accordingly,
291    * using cp in the case of prey species.
292    */
293
```

```
294 static int resweb(realtype tt, N_Vector cc, N_Vector cp,
295                    N_Vector res,  void *rdata)
296 {
297   long int jx, jy, is, yloc, loc, np;
298   realtype *resv, *cpv;
299   UserData webdata;
300
301   webdata = (UserData)rdata;
302
303   cpv = NV_DATA_S(cp);
304   resv = NV_DATA_S(res);
305   np = webdata->np;
306
307   /* Call Fweb to set res to vector of right-hand sides. */
308   Fweb(tt, cc, res, webdata);
309
310   /* Loop over all grid points, setting residual values appropriately
311      for differential or algebraic components.                        */
312
313   for (jy = 0; jy < MY; jy++) {
314     yloc = NSMX * jy;
315     for (jx = 0; jx < MX; jx++) {
316       loc = yloc + NUM_SPECIES * jx;
317       for (is = 0; is < NUM_SPECIES; is++) {
318         if (is < np)
319           resv[loc+is] = cpv[loc+is] - resv[loc+is];
320         else
321           resv[loc+is] = -resv[loc+is];
322       }
323     }
324   }
325
326   return(0);
327
328 }
329
330 /*
331  *------------------------------------------------------------------
332  * PRIVATE FUNCTIONS
333  *------------------------------------------------------------------
334  */
335
336 /*
337  * InitUserData: Load problem constants in webdata (of type UserData).
338  */
339
340 static void InitUserData(UserData webdata)
341 {
342   int i, j, np;
343   realtype *a1,*a2, *a3, *a4, dx2, dy2;
344
345   webdata->mx = MX;
346   webdata->my = MY;
347   webdata->ns = NUM_SPECIES;
348   webdata->np = NPREY;
349   webdata->dx = AX/(MX-1);
350   webdata->dy = AY/(MY-1);
351   webdata->Neq= NEQ;
352
```

```
353     /* Set up the coefficients a and b, and others found in the equations. */
354     np = webdata->np;
355     dx2 = (webdata->dx)*(webdata->dx); dy2 = (webdata->dy)*(webdata->dy);
356
357     for (i = 0; i < np; i++) {
358       a1 = &(acoef[i][np]);
359       a2 = &(acoef[i+np][0]);
360       a3 = &(acoef[i][0]);
361       a4 = &(acoef[i+np][np]);
362       /*  Fill in the portion of acoef in the four quadrants, row by row. */
363       for (j = 0; j < np; j++) {
364         *a1++ =  -GG;
365         *a2++ =   EE;
366         *a3++ = ZERO;
367         *a4++ = ZERO;
368       }
369
370       /* Reset the diagonal elements of acoef to -AA. */
371       acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;
372
373       /* Set coefficients for b and diffusion terms. */
374       bcoef[i] = BB; bcoef[i+np] = -BB;
375       cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
376       coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
377     }
378
379   }
380
381   /*
382    * SetInitialProfiles: Set initial conditions in cc, cp, and id.
383    * A polynomial profile is used for the prey cc values, and a constant
384    * (1.0e5) is loaded as the initial guess for the predator cc values.
385    * The id values are set to 1 for the prey and 0 for the predators.
386    * The prey cp values are set according to the given system, and
387    * the predator cp values are set to zero.
388    */
389
390   static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
391                                  UserData webdata)
392   {
393     long int loc, yloc, is, jx, jy, np;
394     realtype xx, yy, xyfactor, fac;
395     realtype *ccv, *cpv, *idv;
396
397     ccv = NV_DATA_S(cc);
398     cpv = NV_DATA_S(cp);
399     idv = NV_DATA_S(id);
400     np = webdata->np;
401
402     /* Loop over grid, load cc values and id values. */
403     for (jy = 0; jy < MY; jy++) {
404       yy = jy * webdata->dy;
405       yloc = NSMX * jy;
406       for (jx = 0; jx < MX; jx++) {
407         xx = jx * webdata->dx;
408         xyfactor = RCONST(16.0)*xx*(ONE-xx)*yy*(ONE-yy);
409         xyfactor *= xyfactor;
410         loc = yloc + NUM_SPECIES*jx;
411         fac = ONE + ALPHA * xx * yy + BETA * sin(FOURPI*xx) * sin(FOURPI*yy);
```

```
412
413         for (is = 0; is < NUM_SPECIES; is++) {
414           if (is < np) {
415               ccv[loc+is] = RCONST(10.0) + (realtype)(is+1) * xyfactor;
416             idv[loc+is] = ONE;
417           }
418           else {
419             ccv[loc+is] = RCONST(1.0e5);
420             idv[loc+is] = ZERO;
421           }
422         }
423       }
424     }
425
426     /* Set c' for the prey by calling the function Fweb. */
427     Fweb(ZERO, cc, cp, webdata);
428
429     /* Set c' for predators to 0. */
430     for (jy = 0; jy < MY; jy++) {
431       yloc = NSMX * jy;
432       for (jx = 0; jx < MX; jx++) {
433         loc = yloc + NUM_SPECIES * jx;
434         for (is = np; is < NUM_SPECIES; is++) {
435           cpv[loc+is] = ZERO;
436         }
437       }
438     }
439   }
440
441   /*
442    * Print first lines of output (problem description)
443    */
444
445   static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol)
446   {
447     printf("\nidabanx2: Predator-prey DAE serial example problem for IDA \n\n");
448     printf("Number of species ns: %d", NUM_SPECIES);
449     printf("     Mesh dimensions: %d x %d", MX, MY);
450     printf("     System size: %d\n", NEQ);
451   #if defined(SUNDIALS_EXTENDED_PRECISION)
452     printf("Tolerance parameters:   rtol = %Lg   atol = %Lg\n", rtol, atol);
453   #elif defined(SUNDIALS_DOUBLE_PRECISION)
454     printf("Tolerance parameters:   rtol = %lg   atol = %lg\n", rtol, atol);
455   #else
456     printf("Tolerance parameters:   rtol = %g   atol = %g\n", rtol, atol);
457   #endif
458     printf("Linear solver: IDABAND,   Band parameters mu = %ld, ml = %ld\n",mu,ml);
459     printf("CalcIC called to correct initial predator concentrations.\n\n");
460     printf("---------------------------------------------------------\n");
461     printf("  t        bottom-left  top-right");
462     printf("    | nst  k      h\n");
463     printf("---------------------------------------------------------\n\n");
464
465   }
466
467   /*
468    * PrintOutput: Print output values at output time t = tt.
469    * Selected run statistics are printed.  Then values of the concentrations
470    * are printed for the bottom left and top right grid points only.
```

```
 */

static void PrintOutput(void *mem, N_Vector c, realtype t)
{
  int i, kused, flag;
  long int nst;
  realtype *c_bl, *c_tr, hused;

  flag = IDAGetLastOrder(mem, &kused);
  check_flag(&flag, "IDAGetLastOrder", 1);
  flag = IDAGetNumSteps(mem, &nst);
  check_flag(&flag, "IDAGetNumSteps", 1);
  flag = IDAGetLastStep(mem, &hused);
  check_flag(&flag, "IDAGetLastStep", 1);

  c_bl = IJ_Vptr(c,0,0);
  c_tr = IJ_Vptr(c,MX-1,MY-1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%8.2Le %12.4Le %12.4Le    | %3ld  %1d %12.4Le\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("          %12.4Le %12.4Le    |\n",c_bl[i],c_tr[i]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%8.2le %12.4le %12.4le    | %3ld  %1d %12.4le\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("          %12.4le %12.4le    |\n",c_bl[i],c_tr[i]);
#else
  printf("%8.2e %12.4e %12.4e    | %3ld  %1d %12.4e\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("          %12.4e %12.4e    |\n",c_bl[i],c_tr[i]);
#endif

  printf("\n");
}

/*
 * PrintFinalStats: Print final run data contained in iopt.
 */

static void PrintFinalStats(void *mem)
{
  long int nst, nre, nreLS, nni, nje, netf, ncfn;
  int flag;

  flag = IDAGetNumSteps(mem, &nst);
  check_flag(&flag, "IDAGetNumSteps", 1);
  flag = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&flag, "IDAGetNumNonlinSolvIters", 1);
  flag = IDAGetNumResEvals(mem, &nre);
  check_flag(&flag, "IDAGetNumResEvals", 1);
  flag = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&flag, "IDAGetNumErrTestFails", 1);
  flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1);
  flag = IDABandGetNumJacEvals(mem, &nje);
  check_flag(&flag, "IDABandGetNumJacEvals", 1);
```

```
530   flag = IDABandGetNumResEvals(mem, &nreLS);
531   check_flag(&flag, "IDABandGetNumResEvals", 1);
532
533   printf("-------------------------------------------------------------\n");
534   printf("Final run statistics: \n\n");
535   printf("Number of steps                    = %ld\n", nst);
536   printf("Number of residual evaluations     = %ld\n", nre+nreLS);
537   printf("Number of Jacobian evaluations     = %ld\n", nje);
538   printf("Number of nonlinear iterations     = %ld\n", nni);
539   printf("Number of error test failures      = %ld\n", netf);
540   printf("Number of nonlinear conv. failures = %ld\n", ncfn);
541
542 }
543
544 /*
545  * Fweb: Rate function for the food-web problem.
546  * This routine computes the right-hand sides of the system equations,
547  * consisting of the diffusion term and interaction term.
548  * The interaction term is computed by the function WebRates.
549  */
550
551 static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate,
552                  UserData webdata)
553 {
554   long int jx, jy, is, idyu, idyl, idxu, idxl;
555   realtype xx, yy, *cxy, *ratesxy, *cratexy, dcyli, dcyui, dcxli, dcxui;
556
557   /* Loop over grid points, evaluate interaction vector (length ns),
558      form diffusion difference terms, and load crate.                    */
559
560   for (jy = 0; jy < MY; jy++) {
561     yy = (webdata->dy) * jy ;
562     idyu = (jy!=MY-1) ? NSMX : -NSMX;
563     idyl = (jy!= 0  ) ? NSMX : -NSMX;
564
565     for (jx = 0; jx < MX; jx++) {
566       xx = (webdata->dx) * jx;
567       idxu = (jx!= MX-1) ?  NUM_SPECIES : -NUM_SPECIES;
568       idxl = (jx!=  0  ) ?  NUM_SPECIES : -NUM_SPECIES;
569       cxy = IJ_Vptr(cc,jx,jy);
570       ratesxy = IJ_Vptr(webdata->rates,jx,jy);
571       cratexy = IJ_Vptr(crate,jx,jy);
572
573       /* Get interaction vector at this grid point. */
574       WebRates(xx, yy, cxy, ratesxy, webdata);
575
576       /* Loop over species, do differencing, load crate segment. */
577       for (is = 0; is < NUM_SPECIES; is++) {
578
579         /* Differencing in y. */
580         dcyli = *(cxy+is) - *(cxy - idyl + is) ;
581         dcyui = *(cxy + idyu + is) - *(cxy+is);
582
583         /* Differencing in x. */
584         dcxli = *(cxy+is) - *(cxy - idxl + is);
585         dcxui = *(cxy + idxu +is) - *(cxy+is);
586
587         /* Compute the crate values at (xx,yy). */
588         cratexy[is] = coy[is] * (dcyui - dcyli) +
```

```
589             cox[is] * (dcxui - dcxli) + ratesxy[is];
590
591         } /* End is loop */
592       } /* End of jx loop */
593     } /* End of jy loop */
594
595   }
596
597   /*
598    * WebRates: Evaluate reaction rates at a given spatial point.
599    * At a given (x,y), evaluate the array of ns reaction terms R.
600    */
601
602   static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
603                        UserData webdata)
604   {
605     int is;
606     realtype fac;
607
608     for (is = 0; is < NUM_SPECIES; is++)
609       ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);
610
611     fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);
612
613     for (is = 0; is < NUM_SPECIES; is++)
614       ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );
615
616   }
617
618   /*
619    * dotprod: dot product routine for realtype arrays, for use by WebRates.
620    */
621
622   static realtype dotprod(long int size, realtype *x1, realtype *x2)
623   {
624     long int i;
625     realtype *xx1, *xx2, temp = ZERO;
626
627     xx1 = x1; xx2 = x2;
628     for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
629     return(temp);
630
631   }
632
633   /*
634    * Check function return value...
635    *   opt == 0 means SUNDIALS function allocates memory so check if
636    *            returned NULL pointer
637    *   opt == 1 means SUNDIALS function returns a flag so check if
638    *            flag >= 0
639    *   opt == 2 means function allocates memory so check if returned
640    *            NULL pointer
641    */
642
643   static int check_flag(void *flagvalue, char *funcname, int opt)
644   {
645     int *errflag;
646
647     if (opt == 0 && flagvalue == NULL) {
```

```
648       /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
649       fprintf(stderr,
650               "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
651               funcname);
652       return(1);
653     } else if (opt == 1) {
654       /* Check if flag < 0 */
655       errflag = (int *) flagvalue;
656       if (*errflag < 0) {
657         fprintf(stderr,
658                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
659                 funcname, *errflag);
660         return(1);
661       }
662     } else if (opt == 2 && flagvalue == NULL) {
663       /* Check if function returned NULL pointer - no memory allocated */
664       fprintf(stderr,
665               "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
666               funcname);
667       return(1);
668     }
669
670     return(0);
671   }
```

C Listing of idakryx.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.3 $
 * $Date: 2006/02/02 00:34:28 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for IDA: 2D heat equation, serial, GMRES.
 *
 * This example solves a discretized 2D heat equation problem.
 * This version uses the Krylov solver IDASpgmr.
 *
 * The DAE system solved is a spatial discretization of the PDE
 *          du/dt = d^2u/dx^2 + d^2u/dy^2
 * on the unit square. The boundary condition is u = 0 on all edges.
 * Initial conditions are given by u = 16 x (1 - x) y (1 - y). The
 * PDE is treated with central differences on a uniform M x M grid.
 * The values of u at the interior points satisfy ODEs, and
 * equations u = 0 at the boundaries are appended, to form a DAE
 * system of size N = M^2. Here M = 10.
 *
 * The system is solved with IDA/IDAS using the Krylov linear solver
 * IDASPGMR. The preconditioner uses the diagonal elements of the
 * Jacobian only. Routines for preconditioning, required by
 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
 * for all components. Output is taken at t = 0, .01, .02, .04,
 * ..., 10.24. Two cases are run -- with the Gram-Schmidt type
 * being Modified in the first case, and Classical in the second.
 * The second run uses IDAReInit and IDAReInitSpgmr.
 * -----------------------------------------------------------------
 */

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

#include "ida.h"
#include "nvector_serial.h"
#include "ida_spgmr.h"
#include "sundials_types.h"

/* Problem Constants */

#define NOUT  11
#define MGRID 10
#define NEQ   MGRID*MGRID
#define ZERO  RCONST(0.0)
#define ONE   RCONST(1.0)
#define TWO   RCONST(2.0)
#define FOUR  RCONST(4.0)

/* User data type */

typedef struct {
  long int mm;  /* number of grid points */
  realtype dx;
```

```
58    realtype coeff;
59    N_Vector pp;  /* vector of prec. diag. elements */
60  } *UserData;
61
62  /* Prototypes for functions called by IDA */
63
64  int resHeat(realtype tres, N_Vector uu, N_Vector up,
65              N_Vector resval, void *rdata);
66
67  int PsetupHeat(realtype tt,
68                 N_Vector uu, N_Vector up, N_Vector rr,
69                 realtype c_j, void *prec_data,
70                 N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
71
72  int PsolveHeat(realtype tt,
73                 N_Vector uu, N_Vector up, N_Vector rr,
74                 N_Vector rvec, N_Vector zvec,
75                 realtype c_j, realtype delta, void *prec_data,
76                 N_Vector tmp);
77
78  /* Prototypes for private functions */
79
80  static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
81                               N_Vector res);
82  static void PrintHeader(realtype rtol, realtype atol);
83  static void PrintOutput(void *mem, realtype t, N_Vector uu);
84  static int check_flag(void *flagvalue, char *funcname, int opt);
85
86  /*
87   *--------------------------------------------------------------------
88   * MAIN PROGRAM
89   *--------------------------------------------------------------------
90   */
91
92  int main()
93  {
94    void *mem;
95    UserData data;
96    N_Vector uu, up, constraints, res;
97    int ier, iout;
98    realtype rtol, atol, t0, t1, tout, tret;
99    long int netf, ncfn, ncfl;
100
101   mem = NULL;
102   data = NULL;
103   uu = up = constraints = res = NULL;
104
105   /* Allocate N-vectors and the user data structure. */
106
107   uu = N_VNew_Serial(NEQ);
108   if(check_flag((void *)uu, "N_VNew_Serial", 0)) return(1);
109
110   up = N_VNew_Serial(NEQ);
111   if(check_flag((void *)up, "N_VNew_Serial", 0)) return(1);
112
113   res = N_VNew_Serial(NEQ);
114   if(check_flag((void *)res, "N_VNew_Serial", 0)) return(1);
115
116   constraints = N_VNew_Serial(NEQ);
```

```
117     if(check_flag((void *)constraints, "N_VNew_Serial", 0)) return(1);
118
119     data = (UserData) malloc(sizeof *data);
120     data->pp = NULL;
121     if(check_flag((void *)data, "malloc", 2)) return(1);
122
123     /* Assign parameters in the user data structure. */
124
125     data->mm  = MGRID;
126     data->dx = ONE/(MGRID-ONE);
127     data->coeff = ONE/(data->dx * data->dx);
128     data->pp = N_VNew_Serial(NEQ);
129     if(check_flag((void *)data->pp, "N_VNew_Serial", 0)) return(1);
130
131     /* Initialize uu, up. */
132
133     SetInitialProfile(data, uu, up, res);
134
135     /* Set constraints to all 1's for nonnegative solution values. */
136
137     N_VConst(ONE, constraints);
138
139     /* Assign various parameters. */
140
141     t0   = ZERO;
142     t1   = RCONST(0.01);
143     rtol = ZERO;
144     atol = RCONST(1.0e-3);
145
146     /* Call IDACreate and IDAMalloc to initialize solution */
147
148     mem = IDACreate();
149     if(check_flag((void *)mem, "IDACreate", 0)) return(1);
150
151     ier = IDASetRdata(mem, data);
152     if(check_flag(&ier, "IDASetRdata", 1)) return(1);
153
154     ier = IDASetConstraints(mem, constraints);
155     if(check_flag(&ier, "IDASetConstraints", 1)) return(1);
156     N_VDestroy_Serial(constraints);
157
158     ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
159     if(check_flag(&ier, "IDAMalloc", 1)) return(1);
160
161     /* Call IDASpgmr to specify the linear solver. */
162
163     ier = IDASpgmr(mem, 0);
164     if(check_flag(&ier, "IDASpgmr", 1)) return(1);
165
166     ier = IDASpilsSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
167     if(check_flag(&ier, "IDASpilsSetPreconditioner", 1)) return(1);
168
169     /* Print output heading. */
170     PrintHeader(rtol, atol);
171
172     /*
173      * ------------------------------------------------------------------------
174      * CASE I
175      * ------------------------------------------------------------------------
```

```
176      */
177
178    /* Print case number, output table heading, and initial line of table. */
179
180    printf("\n\nCase␣1:␣gsytpe␣=␣MODIFIED_GS\n");
181    printf("\n␣␣␣␣Output␣Summary␣(umax␣=␣max-norm␣of␣solution)␣\n\n");
182    printf("␣␣time␣␣␣␣␣␣umax␣␣␣␣␣␣␣␣k␣␣nst␣␣nni␣␣nje␣␣␣␣nre␣␣␣␣nreLS␣␣␣␣␣h␣␣␣␣␣␣␣npe␣nps\n" );
183    printf("------------------------------------------------------------------------\n");
184
185    /* Loop over output times, call IDASolve, and print results. */
186
187    for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
188      ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
189      if(check_flag(&ier, "IDASolve", 1)) return(1);
190      PrintOutput(mem, tret, uu);
191    }
192
193    /* Print remaining counters. */
194
195    ier = IDAGetNumErrTestFails(mem, &netf);
196    check_flag(&ier, "IDAGetNumErrTestFails", 1);
197
198    ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
199    check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
200
201    ier = IDASpilsGetNumConvFails(mem, &ncfl);
202    check_flag(&ier, "IDASpilsGetNumConvFails", 1);
203
204    printf("\nError␣test␣failures␣␣␣␣␣␣␣␣␣␣␣␣␣=␣%ld\n", netf);
205    printf("Nonlinear␣convergence␣failures␣=␣%ld\n", ncfn);
206    printf("Linear␣convergence␣failures␣␣␣␣=␣%ld\n", ncfl);
207
208    /*
209     * ------------------------------------------------------------------------
210     * CASE II
211     * ------------------------------------------------------------------------
212     */
213
214    /* Re-initialize uu, up. */
215
216    SetInitialProfile(data, uu, up, res);
217
218    /* Re-initialize IDA and IDASPGMR */
219
220    ier = IDAReInit(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
221    if(check_flag(&ier, "IDAReInit", 1)) return(1);
222
223    ier = IDASpilsSetGSType(mem, CLASSICAL_GS);
224    if(check_flag(&ier, "IDASpilsSetGSType",1)) return(1);
225
226    /* Print case number, output table heading, and initial line of table. */
227
228    printf("\n\nCase␣2:␣gstype␣=␣CLASSICAL_GS\n");
229    printf("\n␣␣␣␣Output␣Summary␣(umax␣=␣max-norm␣of␣solution)␣\n\n");
230    printf("␣␣time␣␣␣␣␣␣umax␣␣␣␣␣␣␣␣k␣␣nst␣␣nni␣␣nje␣␣␣␣nre␣␣␣␣nreLS␣␣␣␣␣h␣␣␣␣␣␣␣npe␣nps\n" );
231    printf("------------------------------------------------------------------------\n");
232
233    /* Loop over output times, call IDASolve, and print results. */
234
```

```
235     for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
236       ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
237       if(check_flag(&ier, "IDASolve", 1)) return(1);
238       PrintOutput(mem, tret, uu);
239     }
240 
241     /* Print remaining counters. */
242 
243     ier = IDAGetNumErrTestFails(mem, &netf);
244     check_flag(&ier, "IDAGetNumErrTestFails", 1);
245 
246     ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
247     check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
248 
249     ier = IDASpilsGetNumConvFails(mem, &ncfl);
250     check_flag(&ier, "IDASpilsGetNumConvFails", 1);
251 
252     printf("\nError␣test␣failures␣␣␣␣␣␣␣␣␣␣␣␣=␣%ld\n", netf);
253     printf("Nonlinear␣convergence␣failures␣=␣%ld\n", ncfn);
254     printf("Linear␣convergence␣failures␣␣␣␣=␣%ld\n", ncfl);
255 
256     /* Free Memory */
257 
258     IDAFree(&mem);
259 
260     N_VDestroy_Serial(uu);
261     N_VDestroy_Serial(up);
262     N_VDestroy_Serial(res);
263 
264     N_VDestroy_Serial(data->pp);
265     free(data);
266 
267     return(0);
268   }
269 
270   /*
271    *--------------------------------------------------------------------
272    * FUNCTIONS CALLED BY IDA
273    *--------------------------------------------------------------------
274    */
275 
276   /*
277    * resHeat: heat equation system residual function (user-supplied)
278    * This uses 5-point central differencing on the interior points, and
279    * includes algebraic equations for the boundary values.
280    * So for each interior point, the residual component has the form
281    *    res_i = u'_i - (central difference)_i
282    * while for each boundary point, it is res_i = u_i.
283    */
284 
285   int resHeat(realtype tt,
286               N_Vector uu, N_Vector up, N_Vector rr,
287               void *res_data)
288   {
289     long int i, j, offset, loc, mm;
290     realtype *uu_data, *up_data, *rr_data, coeff, dif1, dif2;
291     UserData data;
292 
293     uu_data = NV_DATA_S(uu);
```

```
294   up_data = NV_DATA_S(up);
295   rr_data = NV_DATA_S(rr);
296
297   data = (UserData) res_data;
298
299   coeff = data->coeff;
300   mm    = data->mm;
301
302   /* Initialize rr to uu, to take care of boundary equations. */
303   N_VScale(ONE, uu, rr);
304
305   /* Loop over interior points; set res = up - (central difference). */
306   for (j = 1; j < MGRID-1; j++) {
307     offset = mm*j;
308     for (i = 1; i < mm-1; i++) {
309       loc = offset + i;
310       dif1 = uu_data[loc-1]  + uu_data[loc+1]  - TWO * uu_data[loc];
311       dif2 = uu_data[loc-mm] + uu_data[loc+mm] - TWO * uu_data[loc];
312       rr_data[loc]= up_data[loc] - coeff * ( dif1 + dif2 );
313     }
314   }
315
316   return(0);
317 }
318
319 /*
320  * PsetupHeat: setup for diagonal preconditioner for idakryx.
321  *
322  * The optional user-supplied functions PsetupHeat and
323  * PsolveHeat together must define the left preconditoner
324  * matrix P approximating the system Jacobian matrix
325  *                   J = dF/du + cj*dF/du'
326  * (where the DAE system is F(t,u,u') = 0), and solve the linear
327  * systems P z = r.   This is done in this case by keeping only
328  * the diagonal elements of the J matrix above, storing them as
329  * inverses in a vector pp, when computed in PsetupHeat, for
330  * subsequent use in PsolveHeat.
331  *
332  * In this instance, only cj and data (user data structure, with
333  * pp etc.) are used from the PsetupdHeat argument list.
334  */
335
336 int PsetupHeat(realtype tt,
337                N_Vector uu, N_Vector up, N_Vector rr,
338                realtype c_j, void *prec_data,
339                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
340 {
341
342   long int i, j, offset, loc, mm;
343   realtype *ppv, pelinv;
344   UserData data;
345
346   data = (UserData) prec_data;
347   ppv = NV_DATA_S(data->pp);
348   mm = data->mm;
349
350   /* Initialize the entire vector to 1., then set the interior points to the
351      correct value for preconditioning. */
352   N_VConst(ONE,data->pp);
```

```
353
354   /* Compute the inverse of the preconditioner diagonal elements. */
355   pelinv = ONE/(c_j + FOUR*data->coeff);
356
357   for (j = 1; j < mm-1; j++) {
358     offset = mm * j;
359     for (i = 1; i < mm-1; i++) {
360       loc = offset + i;
361       ppv[loc] = pelinv;
362     }
363   }
364
365   return(0);
366 }
367
368 /*
369  * PsolveHeat: solve preconditioner linear system.
370  * This routine multiplies the input vector rvec by the vector pp
371  * containing the inverse diagonal Jacobian elements (previously
372  * computed in PrecondHeateq), returning the result in zvec.
373  */
374
375 int PsolveHeat(realtype tt,
376                N_Vector uu, N_Vector up, N_Vector rr,
377                N_Vector rvec, N_Vector zvec,
378                realtype c_j, realtype delta, void *prec_data,
379                N_Vector tmp)
380 {
381   UserData data;
382   data = (UserData) prec_data;
383   N_VProd(data->pp, rvec, zvec);
384   return(0);
385 }
386
387 /*
388  *--------------------------------------------------------------------
389  * PRIVATE FUNCTIONS
390  *--------------------------------------------------------------------
391  */
392
393 /*
394  * SetInitialProfile: routine to initialize u and up vectors.
395  */
396
397 static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
398                              N_Vector res)
399 {
400   long int mm, mm1, i, j, offset, loc;
401   realtype xfact, yfact, *udata, *updata;
402
403   mm = data->mm;
404
405   udata = NV_DATA_S(uu);
406   updata = NV_DATA_S(up);
407
408   /* Initialize uu on all grid points. */
409   mm1 = mm - 1;
410   for (j = 0; j < mm; j++) {
411     yfact = data->dx * j;
```

```
412       offset = mm*j;
413       for (i = 0;i < mm; i++) {
414         xfact = data->dx * i;
415         loc = offset + i;
416         udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
417       }
418     }
419
420     /* Initialize up vector to 0. */
421     N_VConst(ZERO, up);
422
423     /* resHeat sets res to negative of ODE RHS values at interior points. */
424     resHeat(ZERO, uu, up, res, data);
425
426     /* Copy -res into up to get correct interior initial up values. */
427     N_VScale(-ONE, res, up);
428
429     /* Set up at boundary points to zero. */
430     for (j = 0; j < mm; j++) {
431       offset = mm*j;
432       for (i = 0; i < mm; i++) {
433         loc = offset + i;
434         if (j == 0 || j == mm1 || i == 0 || i == mm1 ) updata[loc] = ZERO;
435       }
436     }
437
438     return(0);
439     }
440
441   /*
442    * Print first lines of output (problem description)
443    */
444
445   static void PrintHeader(realtype rtol, realtype atol)
446   {
447     printf("\nidakryx: Heat equation, serial example problem for IDA \n");
448     printf("          Discretized heat equation on 2D unit square. \n");
449     printf("          Zero boundary conditions,");
450     printf(" polynomial initial conditions.\n");
451     printf("          Mesh dimensions: %d x %d", MGRID, MGRID);
452     printf("          Total system size: %d\n\n", NEQ);
453   #if defined(SUNDIALS_EXTENDED_PRECISION)
454     printf("Tolerance parameters:   rtol = %Lg    atol = %Lg\n", rtol, atol);
455   #elif defined(SUNDIALS_DOUBLE_PRECISION)
456     printf("Tolerance parameters:   rtol = %lg    atol = %lg\n", rtol, atol);
457   #else
458     printf("Tolerance parameters:   rtol = %g    atol = %g\n", rtol, atol);
459   #endif
460     printf("Constraints set to force all solution components >= 0. \n");
461     printf("Linear solver: IDASPGMR, preconditioner using diagonal elements. \n");
462   }
463
464   /*
465    * PrintOutput: print max norm of solution and current solver statistics
466    */
467
468   static void PrintOutput(void *mem, realtype t, N_Vector uu)
469   {
470     realtype hused, umax;
```

```
  long int nst, nni, nje, nre, nreLS, nli, npe, nps;
  int kused, ier;

  umax = N_VMaxNorm(uu);

  ier = IDAGetLastOrder(mem, &kused);
  check_flag(&ier, "IDAGetLastOrder", 1);
  ier = IDAGetNumSteps(mem, &nst);
  check_flag(&ier, "IDAGetNumSteps", 1);
  ier = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&ier, "IDAGetNumNonlinSolvIters", 1);
  ier = IDAGetNumResEvals(mem, &nre);
  check_flag(&ier, "IDAGetNumResEvals", 1);
  ier = IDAGetLastStep(mem, &hused);
  check_flag(&ier, "IDAGetLastStep", 1);
  ier = IDASpilsGetNumJtimesEvals(mem, &nje);
  check_flag(&ier, "IDASpilsGetNumJtimesEvals", 1);
  ier = IDASpilsGetNumLinIters(mem, &nli);
  check_flag(&ier, "IDASpilsGetNumLinIters", 1);
  ier = IDASpilsGetNumResEvals(mem, &nreLS);
  check_flag(&ier, "IDASpilsGetNumResEvals", 1);
  ier = IDASpilsGetNumPrecEvals(mem, &npe);
  check_flag(&ier, "IDASpilsGetPrecEvals", 1);
  ier = IDASpilsGetNumPrecSolves(mem, &nps);
  check_flag(&ier, "IDASpilsGetNumPrecSolves", 1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("␣%5.2Lf␣%13.5Le␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2Le␣␣%3ld␣%3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("␣%5.2f␣%13.5le␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2le␣␣%3ld␣%3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
#else
  printf("␣%5.2f␣%13.5e␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2e␣␣%3ld␣%3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
#endif
}

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

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

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

```
530       /* Check if flag < 0 */
531       errflag = (int *) flagvalue;
532       if (*errflag < 0) {
533         fprintf(stderr,
534                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
535                 funcname, *errflag);
536         return(1);
537       }
538     } else if (opt == 2 && flagvalue == NULL) {
539       /* Check if function returned NULL pointer - no memory allocated */
540       fprintf(stderr,
541               "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
542               funcname);
543       return(1);
544     }
545
546     return(0);
547   }
```

D Listing of idakryx1_p.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.6 $
 * $Date: 2006/03/17 16:58:01 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for IDA: 2D heat equation, parallel, GMRES.
 *
 * This example solves a discretized 2D heat equation problem.
 * This version uses the Krylov solver IDASpgmr.
 *
 * The DAE system solved is a spatial discretization of the PDE
 *          du/dt = d^2u/dx^2 + d^2u/dy^2
 * on the unit square. The boundary condition is u = 0 on all edges.
 * Initial conditions are given by u = 16 x (1 - x) y (1 - y).
 * The PDE is treated with central differences on a uniform MX x MY
 * grid. The values of u at the interior points satisfy ODEs, and
 * equations u = 0 at the boundaries are appended, to form a DAE
 * system of size N = MX * MY. Here MX = MY = 10.
 *
 * The system is actually implemented on submeshes, processor by
 * processor, with an MXSUB by MYSUB mesh on each of NPEX * NPEY
 * processors.
 *
 * The system is solved with IDA using the Krylov linear solver
 * IDASPGMR. The preconditioner uses the diagonal elements of the
 * Jacobian only. Routines for preconditioning, required by
 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
 * for all components. Local error testing on the boundary values
 * is suppressed. Output is taken at t = 0, .01, .02, .04,
 * ..., 10.24.
 * -----------------------------------------------------------------
 */

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

#include "ida.h"              /* Main header file */
#include "nvector_parallel.h" /* Definitions of N_Vector and NV_DATA_P */
#include "ida_spgmr.h"        /* Use IDASPGMR linear solver */
#include "sundials_types.h"   /* Definitions of realtype and booleantype */
#include "sundials_math.h"    /* Contains RSqrt routine */

#include "mpi.h"              /* MPI library routines */

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

#define NOUT         11             /* Number of output times */

#define NPEX         2              /* No. PEs in x direction of PE array */
#define NPEY         2              /* No. PEs in y direction of PE array */
                                    /* Total no. PEs = NPEX*NPEY */
```

```
58 #define MXSUB         5              /* No. x points per subgrid */
59 #define MYSUB         5              /* No. y points per subgrid */
60
61 #define MX            (NPEX*MXSUB)   /* MX = number of x mesh points */
62 #define MY            (NPEY*MYSUB)   /* MY = number of y mesh points */
63                                      /* Spatial mesh is MX by MY */
64
65 typedef struct {
66   long int thispe, mx, my, ixsub, jysub, npex, npey, mxsub, mysub;
67   realtype    dx, dy, coeffx, coeffy, coeffxy;
68   realtype    uext[(MXSUB+2)*(MYSUB+2)];
69   N_Vector    pp;    /* vector of diagonal preconditioner elements */
70   MPI_Comm    comm;
71 } *UserData;
72
73 /* User-supplied residual function and supporting routines */
74
75 int resHeat(realtype tt,
76             N_Vector uu, N_Vector up, N_Vector rr,
77             void *res_data);
78
79 static int rescomm(N_Vector uu, N_Vector up, void *res_data);
80
81 static int reslocal(realtype tt, N_Vector uu, N_Vector up,
82                     N_Vector res,  void *res_data);
83
84 static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
85                  long int dsizex, long int dsizey, realtype uarray[]);
86
87 static int BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
88                      long int ixsub, long int jysub,
89                      long int dsizex, long int dsizey,
90                      realtype uext[], realtype buffer[]);
91
92 static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
93                      long int dsizex, realtype uext[], realtype buffer[]);
94
95 /* User-supplied preconditioner routines */
96
97 int PsolveHeat(realtype tt,
98                N_Vector uu, N_Vector up, N_Vector rr,
99                N_Vector rvec, N_Vector zvec,
100               realtype c_j, realtype delta, void *prec_data,
101               N_Vector tmp);
102
103 int PsetupHeat(realtype tt,
104               N_Vector yy, N_Vector yp, N_Vector rr,
105               realtype c_j, void *prec_data,
106               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
107
108 /* Private function to check function return values */
109
110 static int InitUserData(int thispe, MPI_Comm comm, UserData data);
111
112 static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
113                              N_Vector res, UserData data);
114
115 static void PrintHeader(long int Neq, realtype rtol, realtype atol);
116
```

```c
static void PrintOutput(int id, void *mem, realtype t, N_Vector uu);

static void PrintFinalStats(void *mem);

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

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  MPI_Comm comm;
  void *mem;
  UserData data;
  int iout, thispe, ier, npes;
  long int Neq, local_N;
  realtype rtol, atol, t0, t1, tout, tret;
  N_Vector uu, up, constraints, id, res;

  mem = NULL;
  data = NULL;
  uu = up = constraints = id = res = NULL;

  /* Get processor number and total number of pe's. */

  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &thispe);

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

  /* Set local length local_N and global length Neq. */

  local_N = MXSUB*MYSUB;
  Neq     = MX * MY;

  /* Allocate and initialize the data structure and N-vectors. */

  data = (UserData) malloc(sizeof *data);
  data->pp = NULL;
  if(check_flag((void *)data, "malloc", 2, thispe))
    MPI_Abort(comm, 1);

  uu = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)uu, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  up = N_VNew_Parallel(comm, local_N, Neq);
```

```
  if(check_flag((void *)up, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  res = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  constraints = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)constraints, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  id = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  /* An N-vector to hold preconditioner. */
  data->pp = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)data->pp, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  InitUserData(thispe, comm, data);

  /* Initialize the uu, up, id, and res profiles. */

  SetInitialProfile(uu, up, id, res, data);

  /* Set constraints to all 1's for nonnegative solution values. */

  N_VConst(ONE, constraints);

  t0 = ZERO; t1 = RCONST(0.01);

  /* Scalar relative and absolute tolerance. */

  rtol = ZERO;
  atol = RCONST(1.0e-3);

  /* Call IDACreate and IDAMalloc to initialize solution. */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);

  ier = IDASetRdata(mem, data);
  if(check_flag(&ier, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetSuppressAlg(mem, TRUE);
  if(check_flag(&ier, "IDASetSuppressAlg", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetId(mem, id);
  if(check_flag(&ier, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetConstraints(mem, constraints);
  if(check_flag(&ier, "IDASetConstraints", 1, thispe)) MPI_Abort(comm, 1);
  N_VDestroy_Parallel(constraints);

  ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
  if(check_flag(&ier, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);

  /* Call IDASpgmr to specify the linear solver. */
```

```
235
236     ier = IDASpgmr(mem, 0);
237     if(check_flag(&ier, "IDASpgmr", 1, thispe)) MPI_Abort(comm, 1);
238
239     ier = IDASpilsSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
240     if(check_flag(&ier, "IDASpilsSetPreconditioner", 1, thispe)) MPI_Abort(comm, 1);
241
242     /* Print output heading (on processor 0 only) and intial solution  */
243
244     if (thispe == 0) PrintHeader(Neq, rtol, atol);
245     PrintOutput(thispe, mem, t0, uu);
246
247     /* Loop over tout, call IDASolve, print output. */
248
249     for (tout = t1, iout = 1; iout <= NOUT; iout++, tout *= TWO) {
250
251       ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
252       if(check_flag(&ier, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
253
254       PrintOutput(thispe, mem, tret, uu);
255
256     }
257
258     /* Print remaining counters. */
259
260     if (thispe == 0) PrintFinalStats(mem);
261
262     /* Free memory */
263
264     IDAFree(&mem);
265
266     N_VDestroy_Parallel(id);
267     N_VDestroy_Parallel(res);
268     N_VDestroy_Parallel(up);
269     N_VDestroy_Parallel(uu);
270
271     N_VDestroy_Parallel(data->pp);
272     free(data);
273
274     MPI_Finalize();
275
276     return(0);
277
278   }
279
280   /*
281    *--------------------------------------------------------------------
282    * FUNCTIONS CALLED BY IDA
283    *--------------------------------------------------------------------
284    */
285
286   /*
287    * resHeat: heat equation system residual function
288    * This uses 5-point central differencing on the interior points, and
289    * includes algebraic equations for the boundary values.
290    * So for each interior point, the residual component has the form
291    *    res_i = u'_i - (central difference)_i
292    * while for each boundary point, it is res_i = u_i.
293    *
```

```
294   * This parallel implementation uses several supporting routines.
295   * First a call is made to rescomm to do communication of subgrid boundary
296   * data into array uext.  Then reslocal is called to compute the residual
297   * on individual processors and their corresponding domains.  The routines
298   * BSend, BRecvPost, and BREcvWait handle interprocessor communication
299   * of uu required to calculate the residual.
300   */
301
302  int resHeat(realtype tt,
303              N_Vector uu, N_Vector up, N_Vector rr,
304              void *res_data)
305  {
306    int retval;
307
308    /* Call rescomm to do inter-processor communication. */
309    retval = rescomm(uu, up, res_data);
310
311    /* Call reslocal to calculate res. */
312    retval = reslocal(tt, uu, up, rr, res_data);
313
314    return(0);
315
316  }
317
318  /*
319   * PsetupHeat: setup for diagonal preconditioner for heatsk.
320   *
321   * The optional user-supplied functions PsetupHeat and
322   * PsolveHeat together must define the left preconditoner
323   * matrix P approximating the system Jacobian matrix
324   *                   J = dF/du + cj*dF/du'
325   * (where the DAE system is F(t,u,u') = 0), and solve the linear
326   * systems P z = r.   This is done in this case by keeping only
327   * the diagonal elements of the J matrix above, storing them as
328   * inverses in a vector pp, when computed in PsetupHeat, for
329   * subsequent use in PsolveHeat.
330   *
331   * In this instance, only cj and data (user data structure, with
332   * pp etc.) are used from the PsetupHeat argument list.
333   *
334   */
335
336  int PsetupHeat(realtype tt,
337                 N_Vector yy, N_Vector yp, N_Vector rr,
338                 realtype c_j, void *prec_data,
339                 N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
340  {
341    realtype *ppv, pelinv;
342    long int lx, ly, ixbegin, ixend, jybegin, jyend, locu, mxsub, mysub;
343    long int ixsub, jysub, npex, npey;
344    UserData data;
345
346    data = (UserData) prec_data;
347
348    ppv = NV_DATA_P(data->pp);
349    ixsub = data->ixsub;
350    jysub = data->jysub;
351    mxsub = data->mxsub;
352    mysub = data->mysub;
```

```
353     npex  = data->npex;
354     npey  = data->npey;
355
356     /* Initially set all pp elements to one. */
357     N_VConst(ONE, data->pp);
358
359     /* Prepare to loop over subgrid. */
360     ixbegin = 0;
361     ixend   = mxsub-1;
362     jybegin = 0;
363     jyend   = mysub-1;
364     if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
365     if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
366     pelinv = ONE/(c_j + data->coeffxy);
367
368     /* Load the inverse of the preconditioner diagonal elements
369        in loop over all the local subgrid. */
370
371     for (ly = jybegin; ly <=jyend; ly++) {
372       for (lx = ixbegin; lx <= ixend; lx++) {
373         locu  = lx + ly*mxsub;
374         ppv[locu] = pelinv;
375       }
376     }
377
378     return(0);
379
380   }
381
382   /*
383    * PsolveHeat: solve preconditioner linear system.
384    * This routine multiplies the input vector rvec by the vector pp
385    * containing the inverse diagonal Jacobian elements (previously
386    * computed in PsetupHeat), returning the result in zvec.
387    */
388
389   int PsolveHeat(realtype tt,
390                  N_Vector uu, N_Vector up, N_Vector rr,
391                  N_Vector rvec, N_Vector zvec,
392                  realtype c_j, realtype delta, void *prec_data,
393                  N_Vector tmp)
394   {
395     UserData data;
396
397     data = (UserData) prec_data;
398
399     N_VProd(data->pp, rvec, zvec);
400
401     return(0);
402
403   }
404
405   /*
406    *--------------------------------------------------------------------
407    * SUPPORTING FUNCTIONS
408    *--------------------------------------------------------------------
409    */
410
411
```

```
412  /*
413   * rescomm routine.  This routine performs all inter-processor
414   * communication of data in u needed to calculate G.
415   */
416
417  static int rescomm(N_Vector uu, N_Vector up, void *res_data)
418  {
419    UserData data;
420    realtype *uarray, *uext, buffer[2*MYSUB];
421    MPI_Comm comm;
422    long int thispe, ixsub, jysub, mxsub, mysub;
423    MPI_Request request[4];
424
425    data = (UserData) res_data;
426    uarray = NV_DATA_P(uu);
427
428    /* Get comm, thispe, subgrid indices, data sizes, extended array uext. */
429    comm = data->comm;  thispe = data->thispe;
430    ixsub = data->ixsub;   jysub = data->jysub;
431    mxsub = data->mxsub;   mysub = data->mysub;
432    uext = data->uext;
433
434    /* Start receiving boundary data from neighboring PEs. */
435    BRecvPost(comm, request, thispe, ixsub, jysub, mxsub, mysub, uext, buffer);
436
437    /* Send data from boundary of local grid to neighboring PEs. */
438    BSend(comm, thispe, ixsub, jysub, mxsub, mysub, uarray);
439
440    /* Finish receiving boundary data from neighboring PEs. */
441    BRecvWait(request, ixsub, jysub, mxsub, uext, buffer);
442
443    return(0);
444
445  }
446
447  /*
448   * reslocal routine.  Compute res = F(t, uu, up).  This routine assumes
449   * that all inter-processor communication of data needed to calculate F
450   * has already been done, and that this data is in the work array uext.
451   */
452
453  static int reslocal(realtype tt,
454                      N_Vector uu, N_Vector up, N_Vector rr,
455                      void *res_data)
456  {
457    realtype *uext, *uuv, *upv, *resv;
458    realtype termx, termy, termctr;
459    long int lx, ly, offsetu, offsetue, locu, locue;
460    long int ixsub, jysub, mxsub, mxsub2, mysub, npex, npey;
461    long int ixbegin, ixend, jybegin, jyend;
462    UserData data;
463
464    /* Get subgrid indices, array sizes, extended work array uext. */
465
466    data = (UserData) res_data;
467    uext = data->uext;
468    uuv = NV_DATA_P(uu);
469    upv = NV_DATA_P(up);
470    resv = NV_DATA_P(rr);
```

```
471     ixsub = data->ixsub; jysub = data->jysub;
472     mxsub = data->mxsub; mxsub2 = data->mxsub + 2;
473     mysub = data->mysub; npex = data->npex; npey = data->npey;
474
475     /* Initialize all elements of rr to uu. This sets the boundary
476        elements simply without indexing hassles. */
477
478     N_VScale(ONE, uu, rr);
479
480     /* Copy local segment of u vector into the working extended array uext.
481        This completes uext prior to the computation of the rr vector.     */
482
483     offsetu = 0;
484     offsetue = mxsub2 + 1;
485     for (ly = 0; ly < mysub; ly++) {
486       for (lx = 0; lx < mxsub; lx++) uext[offsetue+lx] = uuv[offsetu+lx];
487       offsetu = offsetu + mxsub;
488       offsetue = offsetue + mxsub2;
489     }
490
491     /* Set loop limits for the interior of the local subgrid. */
492
493     ixbegin = 0;
494     ixend   = mxsub-1;
495     jybegin = 0;
496     jyend   = mysub-1;
497     if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
498     if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
499
500     /* Loop over all grid points in local subgrid. */
501
502     for (ly = jybegin; ly <=jyend; ly++) {
503       for (lx = ixbegin; lx <= ixend; lx++) {
504         locu  = lx + ly*mxsub;
505         locue = (lx+1) + (ly+1)*mxsub2;
506         termx = data->coeffx *(uext[locue-1]      + uext[locue+1]);
507         termy = data->coeffy *(uext[locue-mxsub2] + uext[locue+mxsub2]);
508         termctr = data->coeffxy*uext[locue];
509         resv[locu] = upv[locu] - (termx + termy - termctr);
510      }
511     }
512     return(0);
513
514   }
515
516   /*
517    * Routine to send boundary data to neighboring PEs.
518    */
519
520   static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
521                    long int dsizex, long int dsizey, realtype uarray[])
522   {
523     long int ly, offsetu;
524     realtype bufleft[MYSUB], bufright[MYSUB];
525
526     /* If jysub > 0, send data from bottom x-line of u. */
527
528     if (jysub != 0)
529       MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, thispe-NPEX, 0, comm);
```

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

```
589       offsetue = (1 + (MYSUB+1)*(MXSUB+2));
590       MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
591                 thispe+NPEX, 0, comm, &request[1]);
592     }
593
594     /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
595     if (ixsub != 0) {
596       MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
597                 thispe-1, 0, comm, &request[2]);
598     }
599
600     /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
601     if (ixsub != NPEX-1) {
602       MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
603                 thispe+1, 0, comm, &request[3]);
604     }
605
606     return(0);
607
608   }
609
610   /*
611    * Routine to finish receiving boundary data from neighboring PEs.
612    * Notes:
613    *   1) buffer should be able to hold 2*MYSUB realtype entries, should be
614    *      passed to both the BRecvPost and BRecvWait functions, and should not
615    *      be manipulated between the two calls.
616    *   2) request should have four entries, and should be passed in both
617    *      calls also.
618    */
619
620   static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
621                        long int dsizex, realtype uext[], realtype buffer[])
622   {
623     long int ly, dsizex2, offsetue;
624     realtype *bufleft = buffer, *bufright = buffer+MYSUB;
625     MPI_Status status;
626
627     dsizex2 = dsizex + 2;
628
629     /* If jysub > 0, receive data for bottom x-line of uext. */
630     if (jysub != 0)
631       MPI_Wait(&request[0],&status);
632
633     /* If jysub < NPEY-1, receive data for top x-line of uext. */
634     if (jysub != NPEY-1)
635       MPI_Wait(&request[1],&status);
636
637     /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
638     if (ixsub != 0) {
639       MPI_Wait(&request[2],&status);
640
641       /* Copy the buffer to uext. */
642       for (ly = 0; ly < MYSUB; ly++) {
643         offsetue = (ly+1)*dsizex2;
644         uext[offsetue] = bufleft[ly];
645       }
646     }
647
```

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

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetue = (ly+2)*dsizex2 - 1;
      uext[offsetue] = bufright[ly];
    }
  }

  return(0);

}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * InitUserData initializes the user's data block data.
 */

static int InitUserData(int thispe, MPI_Comm comm, UserData data)
{
  data->thispe = thispe;
  data->dx = ONE/(MX-ONE);       /* Assumes a [0,1] interval in x. */
  data->dy = ONE/(MY-ONE);       /* Assumes a [0,1] interval in y. */
  data->coeffx  = ONE/(data->dx * data->dx);
  data->coeffy  = ONE/(data->dy * data->dy);
  data->coeffxy = TWO/(data->dx * data->dx) + TWO/(data->dy * data->dy) ;
  data->jysub   = thispe/NPEX;
  data->ixsub   = thispe - data->jysub * NPEX;
  data->npex    = NPEX;
  data->npey    = NPEY;
  data->mx      = MX;
  data->my      = MY;
  data->mxsub = MXSUB;
  data->mysub = MYSUB;
  data->comm    = comm;
  return(0);

}

/*
 * SetInitialProfile sets the initial values for the problem.
 */

static int SetInitialProfile(N_Vector uu, N_Vector up,  N_Vector id,
                             N_Vector res, UserData data)
{
  long int i, iloc, j, jloc, offset, loc, ixsub, jysub;
  long int ixbegin, ixend, jybegin, jyend;
  realtype xfact, yfact, *udata, *iddata, dx, dy;

  /* Initialize uu. */

```

```
707   udata = NV_DATA_P(uu);
708   iddata = NV_DATA_P(id);
709
710   /* Set mesh spacings and subgrid indices for this PE. */
711   dx = data->dx;
712   dy = data->dy;
713   ixsub = data->ixsub;
714   jysub = data->jysub;
715
716   /* Set beginning and ending locations in the global array corresponding
717      to the portion of that array assigned to this processor. */
718   ixbegin = MXSUB*ixsub;
719   ixend   = MXSUB*(ixsub+1) - 1;
720   jybegin = MYSUB*jysub;
721   jyend   = MYSUB*(jysub+1) - 1;
722
723   /* Loop over the local array, computing the initial profile value.
724      The global indices are (i,j) and the local indices are (iloc,jloc).
725      Also set the id vector to zero for boundary points, one otherwise. */
726
727   N_VConst(ONE,id);
728   for (j = jybegin, jloc = 0; j <= jyend; j++, jloc++) {
729     yfact = data->dy*j;
730     offset= jloc*MXSUB;
731     for (i = ixbegin, iloc = 0; i <= ixend; i++, iloc++) {
732       xfact = data->dx * i;
733       loc = offset + iloc;
734       udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
735       if (i == 0 || i == MX-1 || j == 0 || j == MY-1) iddata[loc] = ZERO;
736     }
737   }
738
739   /* Initialize up. */
740
741   N_VConst(ZERO, up);    /* Initially set up = 0. */
742
743   /* resHeat sets res to negative of ODE RHS values at interior points. */
744   resHeat(ZERO, uu, up, res, data);
745
746   /* Copy -res into up to get correct initial up values. */
747   N_VScale(-ONE, res, up);
748
749   return(0);
750 }
751
752 /*
753  * Print first lines of output and table heading
754  */
755
756 static void PrintHeader(long int Neq, realtype rtol, realtype atol)
757 {
758   printf("\nidakryx1_p:␣Heat␣equation,␣parallel␣example␣problem␣for␣IDA\n");
759   printf("␣␣␣␣␣␣␣␣␣␣␣␣␣Discretized␣heat␣equation␣on␣2D␣unit␣square.\n");
760   printf("␣␣␣␣␣␣␣␣␣␣␣␣␣Zero␣boundary␣conditions,");
761   printf("␣polynomial␣initial␣conditions.\n");
762   printf("␣␣␣␣␣␣␣␣␣␣␣␣␣Mesh␣dimensions:␣%d␣x␣%d", MX, MY);
763   printf("␣␣␣␣␣␣␣␣␣Total␣system␣size:␣%ld\n\n", Neq);
764   printf("Subgrid␣dimensions:␣%d␣x␣%d", MXSUB, MYSUB);
765   printf("␣␣␣␣␣␣␣␣␣Processor␣array:␣%d␣x␣%d\n", NPEX, NPEY);
```

```
766  #if defined(SUNDIALS_EXTENDED_PRECISION)
767    printf("Tolerance␣parameters:␣␣rtol␣=␣%Lg␣␣␣atol␣=␣%Lg\n", rtol, atol);
768  #elif defined(SUNDIALS_DOUBLE_PRECISION)
769    printf("Tolerance␣parameters:␣␣rtol␣=␣%lg␣␣␣atol␣=␣%lg\n", rtol, atol);
770  #else
771    printf("Tolerance␣parameters:␣␣rtol␣=␣%g␣␣␣atol␣=␣%g\n", rtol, atol);
772  #endif
773    printf("Constraints␣set␣to␣force␣all␣solution␣components␣>=␣0.␣\n");
774    printf("SUPPRESSALG␣=␣TRUE␣to␣suppress␣local␣error␣testing␣on␣");
775    printf("all␣boundary␣components.␣\n");
776    printf("Linear␣solver:␣IDASPGMR␣␣");
777    printf("Preconditioner:␣diagonal␣elements␣only.\n");
778
779    /* Print output table heading and initial line of table. */
780    printf("\n␣␣␣Output␣Summary␣(umax␣=␣max-norm␣of␣solution)␣\n\n");
781    printf("␣␣time␣␣␣␣␣umax␣␣␣␣␣␣␣k␣␣nst␣␣nni␣␣nli␣␣␣nre␣␣␣nreLS␣␣␣␣h␣␣␣␣␣␣npe␣nps\n");
782    printf("------------------------------------------------------------------------\n");
783  }
784
785  /*
786   * PrintOutput: print max norm of solution and current solver statistics
787   */
788
789  static void PrintOutput(int id, void *mem, realtype t, N_Vector uu)
790  {
791    realtype hused, umax;
792    long int nst, nni, nje, nre, nreLS, nli, npe, nps;
793    int kused, ier;
794
795    umax = N_VMaxNorm(uu);
796
797    if (id == 0) {
798
799      ier = IDAGetLastOrder(mem, &kused);
800      check_flag(&ier, "IDAGetLastOrder", 1, id);
801      ier = IDAGetNumSteps(mem, &nst);
802      check_flag(&ier, "IDAGetNumSteps", 1, id);
803      ier = IDAGetNumNonlinSolvIters(mem, &nni);
804      check_flag(&ier, "IDAGetNumNonlinSolvIters", 1, id);
805      ier = IDAGetNumResEvals(mem, &nre);
806      check_flag(&ier, "IDAGetNumResEvals", 1, id);
807      ier = IDAGetLastStep(mem, &hused);
808      check_flag(&ier, "IDAGetLastStep", 1, id);
809      ier = IDASpilsGetNumJtimesEvals(mem, &nje);
810      check_flag(&ier, "IDASpilsGetNumJtimesEvals", 1, id);
811      ier = IDASpilsGetNumLinIters(mem, &nli);
812      check_flag(&ier, "IDASpilsGetNumLinIters", 1, id);
813      ier = IDASpilsGetNumResEvals(mem, &nreLS);
814      check_flag(&ier, "IDASpilsGetNumResEvals", 1, id);
815      ier = IDASpilsGetNumPrecEvals(mem, &npe);
816      check_flag(&ier, "IDASpilsGetPrecEvals", 1, id);
817      ier = IDASpilsGetNumPrecSolves(mem, &nps);
818      check_flag(&ier, "IDASpilsGetNumPrecSolves", 1, id);
819
820  #if defined(SUNDIALS_EXTENDED_PRECISION)
821      printf("␣%5.2Lf␣%13.5Le␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2Le␣␣%3ld␣%3ld\n",
822             t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
823  #elif defined(SUNDIALS_DOUBLE_PRECISION)
824      printf("␣%5.2f␣%13.5le␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2le␣␣%3ld␣%3ld\n",
```

```
825             t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
826 #else
827     printf("␣%5.2f␣%13.5e␣␣%d␣␣%3ld␣␣%3ld␣␣%3ld␣␣%4ld␣␣%4ld␣␣%9.2e␣␣%3ld␣%3ld\n",
828             t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
829 #endif
830 
831   }
832 }
833 
834 /*
835  * Print some final integrator statistics
836  */
837 
838 static void PrintFinalStats(void *mem)
839 {
840   long int netf, ncfn, ncfl;
841 
842   IDAGetNumErrTestFails(mem, &netf);
843   IDAGetNumNonlinSolvConvFails(mem, &ncfn);
844   IDASpilsGetNumConvFails(mem, &ncfl);
845 
846   printf("\nError␣test␣failures␣␣␣␣␣␣␣␣␣␣␣␣=␣%ld\n", netf);
847   printf("Nonlinear␣convergence␣failures␣=␣%ld\n", ncfn);
848   printf("Linear␣convergence␣failures␣␣␣␣=␣%ld\n", ncfl);
849 }
850 
851 /*
852  * Check function return value...
853  *   opt == 0 means SUNDIALS function allocates memory so check if
854  *            returned NULL pointer
855  *   opt == 1 means SUNDIALS function returns a flag so check if
856  *            flag >= 0
857  *   opt == 2 means function allocates memory so check if returned
858  *            NULL pointer
859  */
860 
861 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
862 {
863   int *errflag;
864 
865   if (opt == 0 && flagvalue == NULL) {
866     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
867     fprintf(stderr,
868             "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
869             id, funcname);
870     return(1);
871   } else if (opt == 1) {
872     /* Check if flag < 0 */
873     errflag = (int *) flagvalue;
874     if (*errflag < 0) {
875       fprintf(stderr,
876               "\nSUNDIALS_ERROR(%d):␣%s()␣failed␣with␣flag␣=␣%d\n\n",
877               id, funcname, *errflag);
878       return(1);
879     }
880   } else if (opt == 2 && flagvalue == NULL) {
881     /* Check if function returned NULL pointer - no memory allocated */
882     fprintf(stderr,
883             "\nMEMORY_ERROR(%d):␣%s()␣failed␣-␣returned␣NULL␣pointer\n\n",
```

```
884                 id, funcname);
885       return(1);
886     }
887
888     return(0);
889   }
```

E Listing of idakryx2_bbd_p.c

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.5 $
 * $Date: 2006/03/24 15:46:36 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example program for IDA: Food web, parallel, GMRES, IDABBD
 * preconditioner.
 *
 * This example program for IDA uses IDASPGMR as the linear solver.
 * It is written for a parallel computer system and uses the
 * IDABBDPRE band-block-diagonal preconditioner module for the
 * IDASPGMR package. It was originally run on a Sun SPARC cluster
 * and used MPICH.
 *
 * The mathematical problem solved in this example is a DAE system
 * that arises from a system of partial differential equations after
 * spatial discretization. The PDE system is a food web population
 * model, with predator-prey interaction and diffusion on the unit
 * square in two dimensions. The dependent variable vector is:
 *
 *         1   2         ns
 *   c = (c , c ,  ..., c  ) , ns = 2 * np
 *
 * and the PDE's are as follows:
 *
 *     i             i      i
 *   dc /dt = d(i)*(c    + c  )  +  R (x,y,c)   (i = 1,...,np)
 *                   xx     yy       i
 *
 *              i      i
 *   0 = d(i)*(c    + c  )  +  R  (x,y,c)   (i = np+1,...,ns)
 *             xx     yy       i
 *
 *   where the reaction terms R are:
 *
 *                   i             ns         j
 *   R  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                           j=1
 *
 * The number of species is ns = 2 * np, with the first np being
 * prey and the last np being predators. The coefficients a(i,j),
 * b(i), d(i) are:
 *
 *   a(i,i) = -AA  (all i)
 *   a(i,j) = -GG  (i <= np , j >  np)
 *   a(i,j) =  EE  (i >  np,  j <= np)
 *   all other a(i,j) = 0
 *   b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i <= np)
 *   b(i) =-BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i  > np)
 *   d(i) = DPREY  (i <= np)
 *   d(i) = DPRED  (i > np)
 *
 * Note: The above equations are written in 1-based indices,
 * whereas the code has 0-based indices, being written in C.
```

```
 *
 * The various scalar parameters required are set using '#define'
 * statements or directly in routine InitUserData. In this program,
 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
 * normal derivative  =  0.
 *
 * A polynomial in x and y is used to set the initial values of the
 * first np variables (the prey variables) at each x,y location,
 * while initial values for the remaining (predator) variables are
 * set to a flat value, which is corrected by IDACalcIC.
 *
 * The PDEs are discretized by central differencing on a MX by MY
 * mesh, and so the system size Neq is the product
 * MX * MY * NUM_SPECIES. The system is actually implemented on
 * submeshes, processor by processor, with an MXSUB by MYSUB mesh
 * on each of NPEX * NPEY processors.
 *
 * The DAE system is solved by IDA using the IDASPGMR linear solver,
 * in conjunction with the preconditioner module IDABBDPRE. The
 * preconditioner uses a 5-diagonal band-block-diagonal
 * approximation (half-bandwidths = 2). Output is printed at
 * t = 0, .001, .01, .1, .4, .7, 1.
 * -----------------------------------------------------------------
 * References:
 * [1] Peter N. Brown and Alan C. Hindmarsh,
 *     Reduced Storage Matrix Methods in Stiff ODE systems,
 *     Journal of Applied Mathematics and Computation, Vol. 31
 *     (May 1989), pp. 40-91.
 *
 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Using Krylov Methods in the Solution of Large-Scale
 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
 *     (1994), pp. 1467-1488.
 *
 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Consistent Initial Condition Calculation for Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
 *     pp. 1495-1512.
 * -----------------------------------------------------------------
 */

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

#include "ida.h"                 /* Main header file */
#include "nvector_parallel.h"    /* Definitions of N_Vector and NV_DATA_P */
#include "ida_spgmr.h"           /* Use IDASPGMR linear solver */
#include "ida_bbdpre.h"          /* Definitions for the IDABBDPRE prec. */
#include "sundials_smalldense.h" /* definition of denalloc */
#include "sundials_types.h"      /* Definitions of realtype and booleantype */
#include "sundials_math.h"       /* Contains RSqrt routine */

#include "mpi.h"                 /* MPI library routines */

/* Problem Constants */

#define NPREY       1        /* Number of prey (= number of predators). */
#define NUM_SPECIES 2*NPREY
```

```
117
118 #define PI          RCONST(3.1415926535898) /* pi */
119 #define FOURPI      (RCONST(4.0)*PI)        /* 4 pi */
120
121 #define MXSUB       10    /* Number of x mesh points per processor subgrid */
122 #define MYSUB       10    /* Number of y mesh points per processor subgrid */
123 #define NPEX        2     /* Number of subgrids in the x direction */
124 #define NPEY        2     /* Number of subgrids in the y direction */
125 #define MX          (MXSUB*NPEX)      /* MX = number of x mesh points */
126 #define MY          (MYSUB*NPEY)      /* MY = number of y mesh points */
127 #define NSMXSUB     (NUM_SPECIES * MXSUB)
128 #define NEQ         (NUM_SPECIES*MX*MY) /* Number of equations in system */
129 #define AA          RCONST(1.0)     /* Coefficient in above eqns. for a */
130 #define EE          RCONST(10000.) /* Coefficient in above eqns. for a */
131 #define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a */
132 #define BB          RCONST(1.0)     /* Coefficient in above eqns. for b */
133 #define DPREY       RCONST(1.0)     /* Coefficient in above eqns. for d */
134 #define DPRED       RCONST(0.05)    /* Coefficient in above eqns. for d */
135 #define ALPHA       RCONST(50.)     /* Coefficient alpha in above eqns. */
136 #define BETA        RCONST(1000.)   /* Coefficient beta in above eqns. */
137 #define AX          RCONST(1.0)     /* Total range of x variable */
138 #define AY          RCONST(1.0)     /* Total range of y variable */
139 #define RTOL        RCONST(1.e-5)   /*  rtol tolerance */
140 #define ATOL        RCONST(1.e-5)   /*  atol tolerance */
141 #define ZERO        RCONST(0.)      /* 0. */
142 #define ONE         RCONST(1.0)     /* 1. */
143 #define NOUT        6
144 #define TMULT       RCONST(10.0)    /* Multiplier for tout values */
145 #define TADD        RCONST(0.3)     /* Increment for tout values */
146
147 /* User-defined vector accessor macro IJ_Vptr. */
148
149 /*
150  * IJ_Vptr is defined in order to express the underlying 3-d structure of the
151  * dependent variable vector from its underlying 1-d storage (an N_Vector).
152  * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
153  * species index is = 0, x-index ix = i, and y-index jy = j.
154  */
155
156 #define IJ_Vptr(vv,i,j) (&NV_Ith_P(vv, (i)*NUM_SPECIES + (j)*NSMXSUB ))
157
158 /* Type: UserData.  Contains problem constants, preconditioner data, etc. */
159
160 typedef struct {
161   long int ns, np, thispe, npes, ixsub, jysub, npex, npey;
162   long int mxsub, mysub, nsmxsub, nsmxsub2;
163   realtype dx, dy, **acoef;
164   realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES],
165     rhs[NUM_SPECIES], cext[(MXSUB+2)*(MYSUB+2)*NUM_SPECIES];
166   MPI_Comm comm;
167   N_Vector rates;
168   long int n_local;
169 } *UserData;
170
171 /* Prototypes for functions called by the IDA Solver. */
172
173 static int resweb(realtype tt,
174                   N_Vector cc, N_Vector cp, N_Vector rr,
175                   void *res_data);
```

```
176
177 static int reslocal(long int Nlocal, realtype tt,
178                     N_Vector cc, N_Vector cp, N_Vector res,
179                     void *res_data);
180
181 static int rescomm(long int Nlocal, realtype tt,
182                    N_Vector cc, N_Vector cp,
183                    void *res_data);
184
185 /* Prototypes for supporting functions */
186
187 static void BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
188                   long int dsizex, long int dsizey, realtype carray[]);
189
190 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
191                       long int ixsub, long int jysub,
192                       long int dsizex, long int dsizey,
193                       realtype cext[], realtype buffer[]);
194
195 static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
196                       long int dsizex, realtype cext[], realtype buffer[]);
197
198 static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
199                      UserData webdata);
200
201 static realtype dotprod(long int size, realtype *x1, realtype *x2);
202
203 /* Prototypes for private functions */
204
205 static void InitUserData(UserData webdata, int thispe, int npes,
206                          MPI_Comm comm);
207
208 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
209                                N_Vector scrtch, UserData webdata);
210
211 static void PrintHeader(long int SystemSize, int maxl,
212                         long int mudq, long int mldq,
213                         long int mukeep, long int mlkeep,
214                         realtype rtol, realtype atol);
215
216 static void PrintOutput(void *mem, N_Vector cc, realtype time,
217                         UserData webdata, MPI_Comm comm);
218
219 static void PrintFinalStats(void *mem, void *P_data);
220
221 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
222
223 /*
224  *--------------------------------------------------------------------
225  * MAIN PROGRAM
226  *--------------------------------------------------------------------
227  */
228
229 int main(int argc, char *argv[])
230 {
231   MPI_Comm comm;
232   void *mem, *P_data;
233   UserData webdata;
234   long int SystemSize, local_N, mudq, mldq, mukeep, mlkeep;
```

```
235   realtype rtol, atol, t0, tout, tret;
236   N_Vector cc, cp, res, id;
237   int thispe, npes, maxl, iout, retval;
238
239   cc = cp = res = id = NULL;
240   webdata = NULL;
241   mem = P_data = NULL;
242
243   /* Set communicator, and get processor number and total number of PE's. */
244
245   MPI_Init(&argc, &argv);
246   comm = MPI_COMM_WORLD;
247   MPI_Comm_rank(comm, &thispe);
248   MPI_Comm_size(comm, &npes);
249
250   if (npes != NPEX*NPEY) {
251     if (thispe == 0)
252       fprintf(stderr,
253               "\nMPI_ERROR(0): npes = %d not equal to NPEX*NPEY = %d\n",
254               npes, NPEX*NPEY);
255     MPI_Finalize();
256     return(1);
257   }
258
259   /* Set local length (local_N) and global length (SystemSize). */
260
261   local_N = MXSUB*MYSUB*NUM_SPECIES;
262   SystemSize = NEQ;
263
264   /* Set up user data block webdata. */
265
266   webdata = (UserData) malloc(sizeof *webdata);
267   webdata->rates = N_VNew_Parallel(comm, local_N, SystemSize);
268   webdata->acoef = denalloc(NUM_SPECIES);
269
270   InitUserData(webdata, thispe, npes, comm);
271
272   /* Create needed vectors, and load initial values.
273      The vector res is used temporarily only.        */
274
275   cc  = N_VNew_Parallel(comm, local_N, SystemSize);
276   if(check_flag((void *)cc, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
277
278   cp  = N_VNew_Parallel(comm, local_N, SystemSize);
279   if(check_flag((void *)cp, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
280
281   res = N_VNew_Parallel(comm, local_N, SystemSize);
282   if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
283
284   id  = N_VNew_Parallel(comm, local_N, SystemSize);
285   if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
286
287   SetInitialProfiles(cc, cp, id, res, webdata);
288
289   N_VDestroy_Parallel(res);
290
291   /* Set remaining inputs to IDAMalloc. */
292
293   t0 = ZERO;
```

```
294     rtol = RTOL;
295     atol = ATOL;
296
297     /* Call IDACreate and IDAMalloc to initialize solution */
298
299     mem = IDACreate();
300     if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);
301
302     retval = IDASetRdata(mem, webdata);
303     if(check_flag(&retval, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);
304
305     retval = IDASetId(mem, id);
306     if(check_flag(&retval, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);
307
308     retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, rtol, &atol);
309     if(check_flag(&retval, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
310
311     /* Call IDABBDPrecAlloc to initialize the band-block-diagonal preconditioner.
312        The half-bandwidths for the difference quotient evaluation are exact
313        for the system Jacobian, but only a 5-diagonal band matrix is retained. */
314
315     mudq = mldq = NSMXSUB;
316     mukeep = mlkeep = 2;
317     P_data = IDABBDPrecAlloc(mem, local_N, mudq, mldq, mukeep, mlkeep,
318                              ZERO, reslocal, NULL);
319     if(check_flag((void *)P_data, "IDABBDPrecAlloc", 0, thispe)) MPI_Abort(comm, 1);
320
321     /* Call IDABBDSpgmr to specify the IDA linear solver IDASPGMR and specify
322        the preconditioner routines supplied
323        maxl (max. Krylov subspace dim.) is set to 12.   */
324
325     maxl = 12;
326     retval = IDABBDSpgmr(mem, maxl, P_data);
327     if(check_flag(&retval, "IDABBDSpgmr", 1, thispe)) MPI_Abort(comm, 1);
328
329     /* Call IDACalcIC (with default options) to correct the initial values. */
330
331     tout = RCONST(0.001);
332     retval = IDACalcIC(mem, t0, cc, cp, IDA_YA_YDP_INIT, tout);
333     if(check_flag(&retval, "IDACalcIC", 1, thispe)) MPI_Abort(comm, 1);
334
335     /* On PE 0, print heading, basic parameters, initial values. */
336
337     if (thispe == 0) PrintHeader(SystemSize, maxl,
338                                  mudq, mldq, mukeep, mlkeep,
339                                  rtol, atol);
340     PrintOutput(mem, cc, t0, webdata, comm);
341
342     /* Call IDA in tout loop, normal mode, and print selected output. */
343
344     for (iout = 1; iout <= NOUT; iout++) {
345
346       retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
347       if(check_flag(&retval, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
348
349       PrintOutput(mem, cc, tret, webdata, comm);
350
351       if (iout < 3) tout *= TMULT;
352       else          tout += TADD;
```

```
353
354     }
355
356     /* On PE 0, print final set of statistics. */
357
358     if (thispe == 0)  PrintFinalStats(mem, P_data);
359
360     /* Free memory. */
361
362     N_VDestroy_Parallel(cc);
363     N_VDestroy_Parallel(cp);
364     N_VDestroy_Parallel(id);
365
366     IDABBDPrecFree(&P_data);
367
368     IDAFree(&mem);
369
370     denfree(webdata->acoef);
371     N_VDestroy_Parallel(webdata->rates);
372     free(webdata);
373
374     MPI_Finalize();
375
376     return(0);
377   }
378
379   /*
380    *--------------------------------------------------------------------
381    * PRIVATE FUNCTIONS
382    *--------------------------------------------------------------------
383    */
384
385   /*
386    * InitUserData: Load problem constants in webdata (of type UserData).
387    */
388
389   static void InitUserData(UserData webdata, int thispe, int npes,
390                            MPI_Comm comm)
391   {
392     int i, j, np;
393     realtype *a1,*a2, *a3, *a4, dx2, dy2, **acoef, *bcoef, *cox, *coy;
394
395     webdata->jysub = thispe / NPEX;
396     webdata->ixsub = thispe - (webdata->jysub)*NPEX;
397     webdata->mxsub = MXSUB;
398     webdata->mysub = MYSUB;
399     webdata->npex = NPEX;
400     webdata->npey = NPEY;
401     webdata->ns = NUM_SPECIES;
402     webdata->np = NPREY;
403     webdata->dx = AX/(MX-1);
404     webdata->dy = AY/(MY-1);
405     webdata->thispe = thispe;
406     webdata->npes   = npes;
407     webdata->nsmxsub = MXSUB * NUM_SPECIES;
408     webdata->nsmxsub2 = (MXSUB+2)*NUM_SPECIES;
409     webdata->comm = comm;
410     webdata->n_local = MXSUB*MYSUB*NUM_SPECIES;
411
```

```
412   /* Set up the coefficients a and b plus others found in the equations. */
413
414   np = webdata->np;
415   dx2 = (webdata->dx)*(webdata->dx);
416   dy2 = (webdata->dy)*(webdata->dy);
417
418   acoef = webdata->acoef;
419   bcoef = webdata->bcoef;
420   cox = webdata->cox;
421   coy = webdata->coy;
422
423   for (i = 0; i < np; i++) {
424     a1 = &(acoef[i][np]);
425     a2 = &(acoef[i+np][0]);
426     a3 = &(acoef[i][0]);
427     a4 = &(acoef[i+np][np]);
428     /*  Fill in the portion of acoef in the four quadrants, row by row. */
429     for (j = 0; j < np; j++) {
430       *a1++ =  -GG;
431       *a2++ =   EE;
432       *a3++ = ZERO;
433       *a4++ = ZERO;
434     }
435
436     /* Reset the diagonal elements of acoef to -AA. */
437     acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;
438
439     /* Set coefficients for b and diffusion terms. */
440     bcoef[i] = BB; bcoef[i+np] = -BB;
441     cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
442     coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
443   }
444
445 }
446
447 /*
448  * SetInitialProfiles: Set initial conditions in cc, cp, and id.
449  * A polynomial profile is used for the prey cc values, and a constant
450  * (1.0e5) is loaded as the initial guess for the predator cc values.
451  * The id values are set to 1 for the prey and 0 for the predators.
452  * The prey cp values are set according to the given system, and
453  * the predator cp values are set to zero.
454  */
455
456 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
457                                N_Vector res, UserData webdata)
458 {
459   long int ixsub, jysub, mxsub, mysub, nsmxsub, np, ix, jy, is;
460   realtype *cxy, *idxy, *cpxy, dx, dy, xx, yy, xyfactor;
461
462   ixsub = webdata->ixsub;
463   jysub = webdata->jysub;
464   mxsub = webdata->mxsub;
465   mysub = webdata->mxsub;
466   nsmxsub = webdata->nsmxsub;
467   dx = webdata->dx;
468   dy = webdata->dy;
469   np = webdata->np;
470
```

```
471     /* Loop over grid, load cc values and id values. */
472     for (jy = 0; jy < mysub; jy++) {
473       yy = (jy + jysub*mysub) * dy;
474       for (ix = 0; ix < mxsub; ix++) {
475         xx = (ix + ixsub*mxsub) * dx;
476         xyfactor = 16.*xx*(1. - xx)*yy*(1. - yy);
477         xyfactor *= xyfactor;
478 
479         cxy = IJ_Vptr(cc,ix,jy);
480         idxy = IJ_Vptr(id,ix,jy);
481         for (is = 0; is < NUM_SPECIES; is++) {
482           if (is < np) { cxy[is] = RCONST(10.0) + (realtype)(is+1)*xyfactor; idxy[is] = ONE; }
483           else { cxy[is] = 1.0e5; idxy[is] = ZERO; }
484         }
485       }
486     }
487 
488     /* Set c' for the prey by calling the residual function with cp = 0. */
489 
490     N_VConst(ZERO, cp);
491     resweb(ZERO, cc, cp, res, webdata);
492     N_VScale(-ONE, res, cp);
493 
494     /* Set c' for predators to 0. */
495 
496     for (jy = 0; jy < mysub; jy++) {
497       for (ix = 0; ix < mxsub; ix++) {
498         cpxy = IJ_Vptr(cp,ix,jy);
499         for (is = np; is < NUM_SPECIES; is++) cpxy[is] = ZERO;
500       }
501     }
502   }
503 
504   /*
505    * Print first lines of output (problem description)
506    * and table headerr
507    */
508 
509   static void PrintHeader(long int SystemSize, int maxl,
510                           long int mudq, long int mldq,
511                           long int mukeep, long int mlkeep,
512                           realtype rtol, realtype atol)
513   {
514     printf("\nidakryx2_bbd_p: Predator-prey DAE parallel example problem\n\n");
515     printf("Number of species ns: %d\n", NUM_SPECIES);
516     printf("Mesh dimensions:      %d x %d\n", MX, MY);
517     printf("Total system size:    %ld\n",SystemSize);
518     printf("Subgrid dimensions:   %d x %d\n", MXSUB, MYSUB);
519     printf("Processor array:      %d x %d\n", NPEX, NPEY);
520     printf("Tolerance parameters:\n");
521   #if defined(SUNDIALS_EXTENDED_PRECISION)
522     printf(" relative tolerance = %Lg\n", rtol);
523     printf(" absolute tolerance = %Lg\n", atol);
524   #elif defined(SUNDIALS_DOUBLE_PRECISION)
525     printf(" relative tolerance = %lg\n", rtol);
526     printf(" absolute tolerance = %lg\n", atol);
527   #else
528     printf(" relative tolerance = %g\n", rtol);
529     printf(" absolute tolerance = %g\n", atol);
```

```
#endif
  printf("Linear␣solver:␣scaled␣preconditioned␣GMRES␣(IDASPGMR)\n");
  printf("␣max.␣Krylov␣dimension:␣maxl␣=␣%d\n", maxl);
  printf("Preconditioner:␣band-block-diagonal␣(IDABBDPRE)\n");
  printf("␣mudq␣=␣%ld,␣␣mldq␣=␣%ld,␣␣mukeep␣=␣%ld,␣␣mlkeep␣=␣%ld\n",
         mudq, mldq, mukeep, mlkeep);
  printf("CalcIC␣called␣to␣correct␣initial␣predator␣concentrations␣\n\n");
  printf("------------------------------------------------------------\n");
  printf("␣␣t␣␣␣␣␣␣␣␣bottom-left␣␣top-right");
  printf("␣␣␣␣␣|␣nst␣␣k␣␣␣␣␣␣h\n");
  printf("------------------------------------------------------------\n\n");
}


/*
 * PrintOutput: Print output values at output time t = tt.
 * Selected run statistics are printed.  Then values of c1 and c2
 * are printed for the bottom left and top right grid points only.
 */

static void PrintOutput(void *mem, N_Vector cc, realtype tt,
                        UserData webdata, MPI_Comm comm)
{
  MPI_Status status;
  realtype *cdata, clast[2], hused;
  long int nst;
  int i, kused, flag, thispe, npelast, ilast;;

  thispe = webdata->thispe;
  npelast = webdata->npes - 1;
  cdata = NV_DATA_P(cc);

  /* Send conc. at top right mesh point from PE npes-1 to PE 0. */
  if (thispe == npelast) {
    ilast = NUM_SPECIES*MXSUB*MYSUB - 2;
    if (npelast != 0)
      MPI_Send(&cdata[ilast], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else { clast[0] = cdata[ilast]; clast[1] = cdata[ilast+1]; }
  }

  /* On PE 0, receive conc. at top right from PE npes - 1.
     Then print performance data and sampled solution values. */

  if (thispe == 0) {

    if (npelast != 0)
      MPI_Recv(&clast[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);

    flag = IDAGetLastOrder(mem, &kused);
    check_flag(&flag, "IDAGetLastOrder", 1, thispe);
    flag = IDAGetNumSteps(mem, &nst);
    check_flag(&flag, "IDAGetNumSteps", 1, thispe);
    flag = IDAGetLastStep(mem, &hused);
    check_flag(&flag, "IDAGetLastStep", 1, thispe);

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%8.2Le␣%12.4Le␣%12.4Le␣␣␣|␣%3ld␣␣%1d␣%12.4Le\n",
           tt, cdata[0], clast[0], nst, kused, hused);
    for (i=1;i<NUM_SPECIES;i++)
```

```
589         printf("␣␣␣␣␣␣␣␣␣␣%12.4Le␣%12.4Le␣␣␣|\n",cdata[i],clast[i]);
590 #elif defined(SUNDIALS_DOUBLE_PRECISION)
591       printf("%8.2le␣%12.4le␣%12.4le␣␣␣|␣%3ld␣␣%1d␣%12.4le\n",
592            tt, cdata[0], clast[0], nst, kused, hused);
593       for (i=1;i<NUM_SPECIES;i++)
594         printf("␣␣␣␣␣␣␣␣␣␣%12.4le␣%12.4le␣␣␣|\n",cdata[i],clast[i]);
595 #else
596       printf("%8.2e␣%12.4e␣%12.4e␣␣␣|␣%3ld␣␣%1d␣%12.4e\n",
597            tt, cdata[0], clast[0], nst, kused, hused);
598       for (i=1;i<NUM_SPECIES;i++)
599         printf("␣␣␣␣␣␣␣␣␣␣%12.4e␣%12.4e␣␣␣|\n",cdata[i],clast[i]);
600 #endif
601       printf("\n");
602 
603     }
604 
605 }
606 
607 /*
608  * PrintFinalStats: Print final run data contained in iopt.
609  */
610 
611 static void PrintFinalStats(void *mem, void *P_data)
612 {
613   long int nst, nre, nreLS, netf, ncfn, nni, ncfl, nli, npe, nps, nge;
614   int flag;
615 
616   flag = IDAGetNumSteps(mem, &nst);
617   check_flag(&flag, "IDAGetNumSteps", 1, 0);
618   flag = IDAGetNumResEvals(mem, &nre);
619   check_flag(&flag, "IDAGetNumResEvals", 1, 0);
620   flag = IDAGetNumErrTestFails(mem, &netf);
621   check_flag(&flag, "IDAGetNumErrTestFails", 1, 0);
622   flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
623   check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1, 0);
624   flag = IDAGetNumNonlinSolvIters(mem, &nni);
625   check_flag(&flag, "IDAGetNumNonlinSolvIters", 1, 0);
626 
627   flag = IDASpilsGetNumConvFails(mem, &ncfl);
628   check_flag(&flag, "IDASpilsGetNumConvFails", 1, 0);
629   flag = IDASpilsGetNumLinIters(mem, &nli);
630   check_flag(&flag, "IDASpilsGetNumLinIters", 1, 0);
631   flag = IDASpilsGetNumPrecEvals(mem, &npe);
632   check_flag(&flag, "IDASpilsGetNumPrecEvals", 1, 0);
633   flag = IDASpilsGetNumPrecSolves(mem, &nps);
634   check_flag(&flag, "IDASpilsGetNumPrecSolves", 1, 0);
635   flag = IDASpilsGetNumResEvals(mem, &nreLS);
636   check_flag(&flag, "IDASpilsGetNumResEvals", 1, 0);
637 
638   flag = IDABBDPrecGetNumGfnEvals(P_data, &nge);
639   check_flag(&flag, "IDABBDPrecGetNumGfnEvals", 1, 0);
640 
641   printf("------------------------------------------------------------\n");
642   printf("\nFinal␣statistics:␣\n\n");
643 
644   printf("Number␣of␣steps␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣%ld\n", nst);
645   printf("Number␣of␣residual␣evaluations␣␣␣␣␣=␣%ld\n", nre+nreLS);
646   printf("Number␣of␣nonlinear␣iterations␣␣␣␣␣=␣%ld\n", nni);
647   printf("Number␣of␣error␣test␣failures␣␣␣␣␣=␣%ld\n", netf);
```

```
  printf("Number of nonlinear conv. failures = %ld\n\n", ncfn);

  printf("Number of linear iterations         = %ld\n", nli);
  printf("Number of linear conv. failures     = %ld\n\n", ncfl);

  printf("Number of preconditioner setups     = %ld\n", npe);
  printf("Number of preconditioner solves     = %ld\n", nps);
  printf("Number of local residual evals.     = %ld\n", nge);

}

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

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

  if (opt == 0 && flagvalue == NULL) {
    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  }

  return(0);
}

/*
 *-------------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA & SUPPORTING FUNCTIONS
 *-------------------------------------------------------------------
 */

/*
 * resweb: System residual function for predator-prey system.
```

```
 * To compute the residual function F, this routine calls:
 * rescomm, for needed communication, and then
 * reslocal, for computation of the residuals on this processor.
 */

static int resweb(realtype tt,
                  N_Vector cc, N_Vector cp, N_Vector rr,
                  void *res_data)
{
  int retval;
  UserData webdata;
  long int Nlocal;

  webdata = (UserData) res_data;

  Nlocal = webdata->n_local;

  /* Call rescomm to do inter-processor communication. */
  retval = rescomm(Nlocal, tt, cc, cp, res_data);

  /* Call reslocal to calculate the local portion of residual vector. */
  retval = reslocal(Nlocal, tt, cc, cp, rr, res_data);

  return(0);
}

/*
 * rescomm: Communication routine in support of resweb.
 * This routine performs all inter-processor communication of components
 * of the cc vector needed to calculate F, namely the components at all
 * interior subgrid boundaries (ghost cell data).  It loads this data
 * into a work array cext (the local portion of c, extended).
 * The message-passing uses blocking sends, non-blocking receives,
 * and receive-waiting, in routines BRecvPost, BSend, BRecvWait.
 */

static int rescomm(long int Nlocal, realtype tt,
                   N_Vector cc, N_Vector cp,
                   void *res_data)
{

  UserData webdata;
  realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
  long int thispe, ixsub, jysub, nsmxsub, nsmysub;
  MPI_Comm comm;
  MPI_Request request[4];

  webdata = (UserData) res_data;
  cdata = NV_DATA_P(cc);

  /* Get comm, thispe, subgrid indices, data sizes, extended array cext. */

  comm = webdata->comm;
  thispe = webdata->thispe;

  ixsub = webdata->ixsub;
  jysub = webdata->jysub;
  cext = webdata->cext;
  nsmxsub = webdata->nsmxsub;
```

```
766   nsmysub = (webdata->ns)*(webdata->mysub);
767
768   /* Start receiving boundary data from neighboring PEs. */
769
770   BRecvPost(comm, request, thispe, ixsub, jysub, nsmxsub, nsmysub,
771             cext, buffer);
772
773   /* Send data from boundary of local grid to neighboring PEs. */
774
775   BSend(comm, thispe, ixsub, jysub, nsmxsub, nsmysub, cdata);
776
777   /* Finish receiving boundary data from neighboring PEs. */
778
779   BRecvWait(request, ixsub, jysub, nsmxsub, cext, buffer);
780
781   return(0);
782 }
783
784 /*
785  * BRecvPost: Start receiving boundary data from neighboring PEs.
786  * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
787  *     should be passed to both the BRecvPost and BRecvWait functions, and
788  *     should not be manipulated between the two calls.
789  * (2) request should have 4 entries, and is also passed in both calls.
790  */
791
792 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
793                       long int ixsub, long int jysub,
794                       long int dsizex, long int dsizey,
795                       realtype cext[], realtype buffer[])
796 {
797   long int offsetce;
798   /* Have bufleft and bufright use the same buffer. */
799   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
800
801   /* If jysub > 0, receive data for bottom x-line of cext. */
802   if (jysub != 0)
803     MPI_Irecv(&cext[NUM_SPECIES], dsizex, PVEC_REAL_MPI_TYPE,
804               my_pe-NPEX, 0, comm, &request[0]);
805
806   /* If jysub < NPEY-1, receive data for top x-line of cext. */
807   if (jysub != NPEY-1) {
808     offsetce = NUM_SPECIES*(1 + (MYSUB+1)*(MXSUB+2));
809     MPI_Irecv(&cext[offsetce], dsizex, PVEC_REAL_MPI_TYPE,
810               my_pe+NPEX, 0, comm, &request[1]);
811   }
812
813   /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
814   if (ixsub != 0) {
815     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
816               my_pe-1, 0, comm, &request[2]);
817   }
818
819   /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
820   if (ixsub != NPEX-1) {
821     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
822               my_pe+1, 0, comm, &request[3]);
823   }
824
```

```
825 }
826
827 /*
828  * BRecvWait: Finish receiving boundary data from neighboring PEs.
829  * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
830  *     should be passed to both the BRecvPost and BRecvWait functions, and
831  *     should not be manipulated between the two calls.
832  * (2) request should have 4 entries, and is also passed in both calls.
833  */
834
835 static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
836                       long int dsizex, realtype cext[], realtype buffer[])
837 {
838   int i;
839   long int ly, dsizex2, offsetce, offsetbuf;
840   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
841   MPI_Status status;
842
843   dsizex2 = dsizex + 2*NUM_SPECIES;
844
845   /* If jysub > 0, receive data for bottom x-line of cext. */
846   if (jysub != 0)
847     MPI_Wait(&request[0],&status);
848
849   /* If jysub < NPEY-1, receive data for top x-line of cext. */
850   if (jysub != NPEY-1)
851     MPI_Wait(&request[1],&status);
852
853   /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
854   if (ixsub != 0) {
855     MPI_Wait(&request[2],&status);
856
857     /* Copy the buffer to cext */
858     for (ly = 0; ly < MYSUB; ly++) {
859       offsetbuf = ly*NUM_SPECIES;
860       offsetce = (ly+1)*dsizex2;
861       for (i = 0; i < NUM_SPECIES; i++)
862         cext[offsetce+i] = bufleft[offsetbuf+i];
863     }
864   }
865
866   /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
867   if (ixsub != NPEX-1) {
868     MPI_Wait(&request[3],&status);
869
870     /* Copy the buffer to cext */
871     for (ly = 0; ly < MYSUB; ly++) {
872       offsetbuf = ly*NUM_SPECIES;
873       offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
874       for (i = 0; i < NUM_SPECIES; i++)
875         cext[offsetce+i] = bufright[offsetbuf+i];
876     }
877   }
878 }
879
880 /*
881  * BSend: Send boundary data to neighboring PEs.
882  * This routine sends components of cc from internal subgrid boundaries
883  * to the appropriate neighbor PEs.
```

```
884   */
885
886  static void BSend(MPI_Comm comm, long int my_pe, long int ixsub, long int jysub,
887                    long int dsizex, long int dsizey, realtype cdata[])
888  {
889    int i;
890    long int ly, offsetc, offsetbuf;
891    realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];
892
893    /* If jysub > 0, send data from bottom x-line of cc. */
894
895    if (jysub != 0)
896      MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
897
898    /* If jysub < NPEY-1, send data from top x-line of cc. */
899
900    if (jysub != NPEY-1) {
901      offsetc = (MYSUB-1)*dsizex;
902      MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
903    }
904
905    /* If ixsub > 0, send data from left y-line of cc (via bufleft). */
906
907    if (ixsub != 0) {
908      for (ly = 0; ly < MYSUB; ly++) {
909        offsetbuf = ly*NUM_SPECIES;
910        offsetc = ly*dsizex;
911        for (i = 0; i < NUM_SPECIES; i++)
912          bufleft[offsetbuf+i] = cdata[offsetc+i];
913      }
914      MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
915    }
916
917    /* If ixsub < NPEX-1, send data from right y-line of cc (via bufright). */
918
919    if (ixsub != NPEX-1) {
920      for (ly = 0; ly < MYSUB; ly++) {
921        offsetbuf = ly*NUM_SPECIES;
922        offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
923        for (i = 0; i < NUM_SPECIES; i++)
924          bufright[offsetbuf+i] = cdata[offsetc+i];
925      }
926      MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
927    }
928  }
929
930  /* Define lines are for ease of readability in the following functions. */
931
932  #define mxsub      (webdata->mxsub)
933  #define mysub      (webdata->mysub)
934  #define npex       (webdata->npex)
935  #define npey       (webdata->npey)
936  #define ixsub      (webdata->ixsub)
937  #define jysub      (webdata->jysub)
938  #define nsmxsub    (webdata->nsmxsub)
939  #define nsmxsub2   (webdata->nsmxsub2)
940  #define np         (webdata->np)
941  #define dx         (webdata->dx)
942  #define dy         (webdata->dy)
```

```
943 #define cox         (webdata->cox)
944 #define coy         (webdata->coy)
945 #define rhs         (webdata->rhs)
946 #define cext        (webdata->cext)
947 #define rates       (webdata->rates)
948 #define ns          (webdata->ns)
949 #define acoef       (webdata->acoef)
950 #define bcoef       (webdata->bcoef)
951
952 /*
953  * reslocal: Compute res = F(t,cc,cp).
954  * This routine assumes that all inter-processor communication of data
955  * needed to calculate F has already been done.  Components at interior
956  * subgrid boundaries are assumed to be in the work array cext.
957  * The local portion of the cc vector is first copied into cext.
958  * The exterior Neumann boundary conditions are explicitly handled here
959  * by copying data from the first interior mesh line to the ghost cell
960  * locations in cext.  Then the reaction and diffusion terms are
961  * evaluated in terms of the cext array, and the residuals are formed.
962  * The reaction terms are saved separately in the vector webdata->rates
963  * for use by the preconditioner setup routine.
964  */
965
966 static int reslocal(long int Nlocal, realtype tt,
967                     N_Vector cc, N_Vector cp, N_Vector rr,
968                     void *res_data)
969 {
970   realtype *cdata, *ratesxy, *cpxy, *resxy,
971     xx, yy, dcyli, dcyui, dcxli, dcxui;
972   long int ix, jy, is, i, locc, ylocce, locce;
973   UserData webdata;
974
975   webdata = (UserData) res_data;
976
977   /* Get data pointers, subgrid data, array sizes, work array cext. */
978
979   cdata = NV_DATA_P(cc);
980
981   /* Copy local segment of cc vector into the working extended array cext. */
982
983   locc = 0;
984   locce = nsmxsub2 + NUM_SPECIES;
985   for (jy = 0; jy < mysub; jy++) {
986     for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
987     locc = locc + nsmxsub;
988     locce = locce + nsmxsub2;
989   }
990
991   /* To facilitate homogeneous Neumann boundary conditions, when this is
992      a boundary PE, copy data from the first interior mesh line of cc to cext. */
993
994   /* If jysub = 0, copy x-line 2 of cc to cext. */
995   if (jysub == 0)
996     { for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i]; }
997
998   /* If jysub = npey-1, copy x-line mysub-1 of cc to cext. */
999   if (jysub == npey-1) {
1000     locc = (mysub-2)*nsmxsub;
1001     locce = (mysub+1)*nsmxsub2 + NUM_SPECIES;
```

```
1002      for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
1003    }
1004
1005    /* If ixsub = 0, copy y-line 2 of cc to cext. */
1006    if (ixsub == 0) {
1007      for (jy = 0; jy < mysub; jy++) {
1008        locc = jy*nsmxsub + NUM_SPECIES;
1009        locce = (jy+1)*nsmxsub2;
1010        for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
1011      }
1012    }
1013
1014    /* If ixsub = npex-1, copy y-line mxsub-1 of cc to cext. */
1015    if (ixsub == npex-1) {
1016      for (jy = 0; jy < mysub; jy++) {
1017        locc  = (jy+1)*nsmxsub - 2*NUM_SPECIES;
1018        locce = (jy+2)*nsmxsub2 - NUM_SPECIES;
1019        for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
1020      }
1021    }
1022
1023    /* Loop over all grid points, setting local array rates to right-hand sides.
1024       Then set rr values appropriately for prey/predator components of F. */
1025
1026    for (jy = 0; jy < mysub; jy++) {
1027      ylocce = (jy+1)*nsmxsub2;
1028      yy     = (jy+jysub*mysub)*dy;
1029
1030      for (ix = 0; ix < mxsub; ix++) {
1031        locce = ylocce + (ix+1)*NUM_SPECIES;
1032        xx = (ix + ixsub*mxsub)*dx;
1033
1034        ratesxy = IJ_Vptr(rates,ix,jy);
1035        WebRates(xx, yy, &(cext[locce]), ratesxy, webdata);
1036
1037        resxy = IJ_Vptr(rr,ix,jy);
1038        cpxy = IJ_Vptr(cp,ix,jy);
1039
1040        for (is = 0; is < NUM_SPECIES; is++) {
1041          dcyli = cext[locce+is]          - cext[locce+is-nsmxsub2];
1042          dcyui = cext[locce+is+nsmxsub2] - cext[locce+is];
1043
1044          dcxli = cext[locce+is]              - cext[locce+is-NUM_SPECIES];
1045          dcxui = cext[locce+is+NUM_SPECIES] - cext[locce+is];
1046
1047          rhs[is] = cox[is]*(dcxui-dcxli) + coy[is]*(dcyui-dcyli) + ratesxy[is];
1048
1049          if (is < np) resxy[is] = cpxy[is] - rhs[is];
1050          else         resxy[is] =          - rhs[is];
1051
1052        }
1053      }
1054    }
1055
1056    return(0);
1057  }
1058
1059  /*
1060   * WebRates: Evaluate reaction rates at a given spatial point.
```

```
1061   * At a given (x,y), evaluate the array of ns reaction terms R.
1062   */
1063
1064  static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
1065                       UserData webdata)
1066  {
1067    int is;
1068    realtype fac;
1069
1070    for (is = 0; is < NUM_SPECIES; is++)
1071      ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);
1072
1073    fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);
1074
1075    for (is = 0; is < NUM_SPECIES; is++)
1076      ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );
1077
1078  }
1079
1080  /*
1081   * dotprod: dot product routine for realtype arrays, for use by WebRates.
1082   */
1083
1084  static realtype dotprod(long int size, realtype *x1, realtype *x2)
1085  {
1086    long int i;
1087    realtype *xx1, *xx2, temp = ZERO;
1088
1089    xx1 = x1;
1090    xx2 = x2;
1091    for (i = 0; i < size; i++)
1092      temp += (*xx1++) * (*xx2++);
1093
1094    return(temp);
1095  }
1096
```

F Listing of fidadenx.f

```
c     ----------------------------------------------------------------
c     $Revision: 1.5 $
c     $Date: 2006/03/24 00:28:51 $
c     ----------------------------------------------------------------
c     This simple example problem for FIDA, due to Robertson, is from
c     chemical kinetics, and consists of the following three equations:
c
c          dy1/dt = -.04*y1 + 1.e4*y2*y3
c          dy2/dt =  .04*y1 - 1.e4*y2*y3 - 3.e7*y2**2
c             0   = y1 + y2 + y3 - 1
c
c     on the interval from t = 0.0 to t = 4.e10, with initial
c     conditions: y1 = 1, y2 = y3 = 0.
c
c     While integrating the system, we also use the rootfinding feature
c     to find the points at which y1 = 1.e-4 or at which y3 = 0.01.
c
c     The problem is solved using a dense linear solver, with a
c     user-supplied Jacobian. Output is printed at
c     t = .4, 4, 40, ..., 4e10.
c     ----------------------------------------------------------------
c
      program fidadenx
c
      implicit none
c
      integer ier, ierroot, info(2)
      integer*4 iout(25), ipar
      double precision rout(10), rpar
c
      integer iatol, nout, jout, itask
      integer nst, kused, hused
      integer*4 neq, i
      double precision t0, t1, rtol, tout, tret
      double precision y(3), yp(3), atol(3)
c
      data nst/3/, kused/9/, hused/2/
c
c Initialize variables
c
      neq = 3
      nout = 12
      rtol = 1.0d-4
      t0 = 0.0d0
      t1 = 0.4d0
      iatol = 2
      itask = 1
c
      y(1) = 1.0d0
      y(2) = 0.0d0
      y(3) = 0.0d0
c
      yp(1) = -0.04d0
      yp(2) = 0.04d0
      yp(3) = 0.0d0
c
      atol(1) = 1.0d-6
```

```
      atol(2) = 1.0d-10
      atol(3) = 1.0d-6
c
c Initialize IDA vector environment
c
      call fnvinits(2, neq, ier)
      if (ier .ne. 0) then
         write(6,10) ier
 10      format(///' SUNDIALS_ERROR: FNVINITS returned IER = ', i5)
         stop
      endif
c
      call fidamalloc(t0, y, yp, iatol, rtol, atol,
     &                iout, rout, ipar, rpar, ier)
      if (ier .ne. 0) then
         write(6,20) ier
 20      format(///' SUNDIALS_ERROR: FIDAMALLOC returned IER = ', i5)
         stop
      endif
c
c Initialize rootfinding problem

      call fidarootinit(2, ier)
      if (ier .ne. 0) then
         write(6,25) ier
 25      format(///' SUNDIALS_ERROR: FIDAROOTINIT returned IER = ', i5)
         call fidafree
         stop
      endif
c
c Attach dense linear solver
c
      call fidadense(neq, ier)
      call fidadensesetjac(1, ier)
c
c Print header
c
      call prntintro(rtol, atol, y)
c
      tout = t1
c
c
      jout = 1
      do while(jout .le. nout)
c
        call fidasolve(tout, tret, y, yp, itask, ier)
c
        write(6,40) tret, (y(i), i = 1,3), iout(nst), iout(kused),
     &              rout(hused)
 40     format(e10.4, 3(1x,e12.4), i5, i3, e12.4)
c
        if (ier .lt. 0) then
           write(6,50) ier, iout(15)
 50        format(///' SUNDIALS_ERROR: FIDASOLVE returned IER = ',i5,/,
     1            '                  Linear Solver returned IER = ',i5)
           call fidarootfree
           call fidafree
           stop
        endif
```

```
c
        if (ier .eq. 2) then
          call fidarootinfo(2, info, ierroot)
          if (ierroot .lt. 0) then
            write(6,55) ier
 55         format(///' SUNDIALS_ERROR: FIDAROOTINFO returned IER = ',
     1             i5)
            call fidarootfree
            call fidafree
            stop
          endif
          write(6,60) (info(i), i = 1,2)
 60       format(5x, 'Above is a root, INFO() = ', 2i3)
        endif
c
        if (ier .eq. 0) then
           tout = tout * 10.0d0
           jout = jout + 1
        endif
c
      ENDDO
c
c Print final statistics
c
      call prntstats(iout)
c
c Free IDA memory
c
      call fidarootfree
      call fidafree
c
      stop
      end
c
c ==========
c
      subroutine fidaresfun(tres, y, yp, res, ipar, rpar, reserr)
c
      implicit none
c
      integer reserr
      integer*4 ipar(*)
      double precision tres, rpar(*)
      double precision y(*), yp(*), res(*)
c
      res(1) = -0.04d0*y(1)+1.0d4*y(2)*y(3)
      res(2) = -res(1)-3.0d7*y(2)*y(2)-yp(2)
      res(1) = res(1)-yp(1)
      res(3) = y(1)+y(2)+y(3)-1.0d0
c
      reserr = 0
c
      return
      end
c
c ==========
c
      subroutine fidadjac(neq, t, y, yp, r, jac, cj, ewt, h,
     1                    ipar, rpar, wk1, wk2, wk3, djacerr)
```

```
c
      implicit none
c
      integer*4 neq, ipar(*)
      integer djacerr
      double precision t, h, cj, rpar(*)
      double precision y(*), yp(*), r(*), ewt(*), jac(neq,neq)
      double precision wk1(*), wk2(*), wk3(*)
c
      jac(1,1) = -0.04d0-cj
      jac(2,1) = 0.04d0
      jac(3,1) = 1.0d0
      jac(1,2) = 1.0d4*y(3)
      jac(2,2) = -1.0d4*y(3)-6.0d7*y(2)-cj
      jac(3,2) = 1.0d0
      jac(1,3) = 1.0d4*y(2)
      jac(2,3) = -1.0d4*y(2)
      jac(3,3) = 1.0d0
c
      djacerr = 0

      return
      end
c
c ==========
c
      subroutine fidarootfn(t, y, yp, g, ipar, rpar, ier)
c Fortran routine for rootfinding
      implicit none
c
      INTEGER*4 ipar(*), ier
      double precision t, y(*), yp(*), g(*), rpar(*)
c
      g(1) = y(1) - 1.0d-4
      g(2) = y(3) - 1.0d-2

      ier = 0

      return
      end
c
c ==========
c
      subroutine prntintro(rtol, atol, y)
c
      implicit none
c
      integer*4 i
      double precision rtol, atol(*), y(*)
c
      write(6,60) rtol, (atol(i), i = 1,3), (y(i), i = 1,3)
 60   format(/'fidadenx: Robertson kinetics DAE serial example',
     &       'problem for IDA', /,'          Three equation chemical',
     &       'kinetics problem.', //,
     &       'Tolerance parameters:  rtol = ', e8.2,
     &       '   atol = ', 3(1x,e8.2), /,
     &       'Initial conditions y0 = (', 3(1x,e8.2), ')', //,
     &       '  t            y1           y2           y3        nst',
     &       '  k    h')
```

```
c
      return
      end
c
c ==========
c
      subroutine prntstats(iout)
c
      implicit none
c
      integer*4 iout(25)
      integer nst, reseval, jaceval, nni, ncf, netf, nge
c
      data nst/3/, reseval/4/, jaceval/17/, nni/7/, netf/5/,
     &     ncf/6/, nge/12/
c
      write(6,70) iout(nst), iout(reseval), iout(jaceval),
     &            iout(nni), iout(netf), iout(ncf), iout(nge)
 70   format(/'Final Run Statistics:', //,
     &        'Number of steps                    = ', i3, /,
     &        'Number of residual evaluations     = ', i3, /,
     &        'Number of Jacobian evaluations     = ', i3, /,
     &        'Number of nonlinear iterations     = ', i3, /,
     &        'Number of error test failures      = ', i3, /,
     &        'Number of nonlinear conv. failures = ', i3, /,
     &        'Number of root function evals.     = ', i3)
c
      return
      end
```

G Listing of fidakryx_bbd_p.f

```
c     ----------------------------------------------------------------
c     $Revision: 1.4 $
c     $Date: 2006/03/24 15:46:37 $
c     ----------------------------------------------------------------
c     Example problem for FIDA: 2D heat equation, parallel, GMRES,
c     IDABBDPRE.
c
c     This example solves a discretized 2D heat equation problem.
c     This version uses the Krylov solver IDASPGMR and BBD
c     preconditioning.
c
c     The DAE system solved is a spatial discretization of the PDE
c                du/dt = d^2u/dx^2 + d^2u/dy^2
c     on the unit square. The boundary condition is u = 0 on all edges.
c     Initial conditions are given by u = 16 x (1 - x) y (1 - y). The
c     PDE is treated with central differences on a uniform MX x MY
c     grid. The values of u at the interior points satisfy ODEs, and
c     equations u = 0 at the boundaries are appended, to form a DAE
c     system of size N = MX * MY. Here MX = MY = 10.
c
c     The system is actually implemented on submeshes, processor by
c     processor, with an MXSUB by MYSUB mesh on each of NPEX * NPEY
c     processors.
c
c     The system is solved with FIDA using the Krylov linear solver
c     IDASPGMR in conjunction with the preconditioner module IDABBDPRE.
c     The preconditioner uses a tridiagonal approximation
c     (half-bandwidths = 1). The constraints u >= 0 are posed for all
c     components. Local error testing on the boundary values is
c     suppressed. Output is taken at t = 0, .01, .02, .04, ..., 10.24.
c     ----------------------------------------------------------------
c
      program fidakryx_bbd_p
c
      include "mpif.h"
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      integer*4 mudq, mldq, mukeep, mlkeep
      integer*4 iout(25), ipar
      double precision rout(10), rpar
      integer nout, ier
      parameter (nout = 11)
      integer npes, inopt, maxl, gstype, maxrs, itask, iatol
      double precision t0, t1, tout, tret, dqrely, eplifac, dqincfac
      double precision atol, rtol
```

```
58        double precision constr(nlocalg), uu(nlocalg), up(nlocalg)
59        double precision res(nlocalg), id(nlocalg)
60  c
61        data atol/1.0d-3/, rtol/0.0d0/
62  c
63        common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
64       &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
65       &              ixsub, jysub, thispe
66  c
67  c Initialize variables
68  c
69        npex = 2
70        npey = 2
71        mxsub = 5
72        mysub = 5
73        mx = npex*mxsub
74        my = npey*mysub
75        neq = mx*my
76        nlocal = mxsub*mysub
77        inopt = 1
78        t0 = 0.0d0
79        t1 = 0.01d0
80        mudq = mxsub
81        mldq = mxsub
82        mukeep = 1
83        mlkeep = 1
84        dqrely = 0.0d0
85        maxl = 0
86        gstype = 0
87        maxrs = 0
88        eplifac = 0.0d0
89        dqincfac = 0.0d0
90        itask = 1
91        iatol = 1
92  c
93  c Initialize MPI environment
94  c
95        call mpi_init(ier)
96        if (ier .ne. 0) then
97           write(*,2) ier
98   2       format(///' MPI_ERROR: MPI_INIT returned IER = ', i5)
99           stop
100       endif
101 c
102       call mpi_comm_size(mpi_comm_world, npes, ier)
103       if (ier .ne. 0) then
104          write(*,3) ier
105  3       format(///' MPI_ERROR: MPI_COMM_SIZE returned IER = ', i5)
106          call mpi_abort(mpi_comm_world, 1, ier)
107          stop
108       endif
109 c
110       call mpi_comm_rank(mpi_comm_world, thispe, ier)
111       if (ier .ne. 0) then
112          write(*,4) ier
113  4       format(///' MPI_ERROR: MPI_COMM_RANK returned IER = ', i5)
114          call mpi_abort(mpi_comm_world, 1, ier)
115          stop
116       endif
```

```
c
      if (npes .ne. npex*npey) then
         if (thispe .eq. 0) then
            write(*,5) npes, npex*npey
 5          format(///' MPI_ERROR: npes = ', i5, ' is not equal to ',
     &                 'NPEX*NPEY = ', i5)
            call mpi_finalize(ier)
            stop
         endif
      endif
c
      call fnvinitp(mpi_comm_world, 2, nlocal, neq, ier)
      if (ier .ne. 0) then
         write(*,6) ier
 6       format(///' SUNDIALS_ERROR: FNVINITP returned IER = ', i5)
         call mpi_finalize(ier)
         stop
      endif
c
      jysub = int(thispe/npex)
      ixsub = thispe-jysub*npex
c
c Initialize problem data
c
      call setinitprofile(uu, up, id, res, constr, ipar, rpar)
c
c Initialize IDA environment
c
      call fidamalloc(t0, uu, up, iatol, rtol, atol,
     &     iout, rout, ipar, rpar, ier)
      if (ier .ne. 0) then
         write(*,7) ier
 7       format(///' SUNDIALS_ERROR: FIDAMALLOC returned IER = ', i5)
         call mpi_abort(mpi_comm_world, 1, ier)
         stop
      endif
c
c     Set optional inputs
c
      call fidasetiin('SUPPRESS_ALG', 1, ier)
      call fidasetvin('ID_VEC', id, ier)
      call fidasetvin('CONSTR_VEC', constr, ier)

c
c Initialize and attach BBDSPGMR module
c
      call fidabbdinit(nlocal, mudq, mldq, mukeep, mlkeep, dqrely, ier)
      if (ier .ne. 0) then
         write(*,8) ier
 8       format(///' SUNDIALS_ERROR: FIDABBDINIT returned IER = ', i5)
         call mpi_abort(mpi_comm_world, 1, ier)
         stop
      endif
c
      call fidabbdspgmr(maxl, gstype, maxrs, eplifac, dqincfac, ier)
      if (ier .ne. 0) then
         write(*,9) ier
 9       format(///' SUNDIALS_ERROR: FIDABBDSPGMR returned IER = ', i5)
         call mpi_abort(mpi_comm_world, 1, ier)
```

```
176           stop
177        endif
178  c
179  c Print header
180  c
181        if (thispe .eq. 0) then
182           call prntintro(rtol, atol)
183           call prntcase(1, mudq, mukeep)
184        endif
185  c
186        tout = t1
187        do 10 jout = 1, nout
188  c
189           call fidasolve(tout, tret, uu, up, itask, ier)
190  c
191           call prntoutput(tret, uu, iout, rout)
192  c
193           if (ier .ne. 0) then
194              write(*,11) ier
195   11         format(///' SUNDIALS_ERROR: FIDASOLVE returned IER = ', i5)
196              call fidafree
197              stop
198           endif
199  c
200           tout = tout*2.0d0
201  c
202    10    continue
203  c
204  c Print statistics
205  c
206        if (thispe .eq. 0) then
207           call prntfinalstats(iout)
208        endif
209  c
210  c Reinitialize variables and data for second problem
211  c
212        mudq = 1
213        mldq = 1
214        jysub = thispe/npex
215        ixsub = thispe-jysub*npex
216  c
217        call setinitprofile(uu, up, id, res, constr, ipar, rpar)
218  c
219        call fidareinit(t0, uu, up, iatol, rtol, atol, ier)
220        if (ier .ne. 0) then
221           write(*,33) ier
222    33     format(///' SUNDIALS_ERROR: FIDAREINIT returned IER = ', i5)
223        endif
224  c
225        call fidabbdreinit(nlocal, mudq, mldq, dqrely, ier)
226        if (ier .ne. 0) then
227           write(*,34) ier
228    34     format(///' SUNDIALS_ERROR: FIDABBDREINIT returned IER = ', i5)
229           call fidafree
230           stop
231        endif
232  c
233  c Print header
234  c
```

```
      if (thispe .eq. 0) then
         call prntcase(2, mudq, mukeep)
      endif
c
      tout = t1
      do 12 jout = 1, nout
c
         call fidasolve(tout, tret, uu, up, itask, ier)
c
         call prntoutput(tret, uu, iout, rout)
c
         if (ier .ne. 0) then
            write(*,13) ier
 13         format(///' SUNDIALS_ERROR: FIDASOLVE returned IER = ', i5)
            call fidafree
            stop
         endif
c
         tout = tout*2.0d0
c
 12   continue
c
c Print statistics
c
      if (thispe .eq. 0) then
         call prntfinalstats(iout)
      endif
c
c Free memory
c
      call fidabbdfree
      call fidafree
c
      call mpi_finalize(ier)
c
      stop
      end
c
c ==========
c
      subroutine setinitprofile(uu, up, id, res, constr, ipar, rpar)
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub, ipar(*)
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      integer*4 i, iloc, j, jloc, offset, loc
      integer*4 ixbegin, ixend, jybegin, jyend
      integer reserr
      double precision xfact, yfact
```

```
294         double precision uu(*), up(*), id(*), res(*), constr(*)
295   c
296         common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
297        &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
298        &              ixsub, jysub, thispe
299   c
300   c Initialize variables
301   c
302         dx = 1.0d0/dble(mx-1)
303         dy = 1.0d0/dble(my-1)
304         coeffx = 1.0d0/(dx*dx)
305         coeffy = 1.0d0/(dy*dy)
306         coeffxy = 2.0d0/(dx*dx)+2.0d0/(dy*dy)
307         ixbegin = mxsub*ixsub
308         ixend = mxsub*(ixsub+1)-1
309         jybegin = mysub*jysub
310         jyend = mysub*(jysub+1)-1
311   c
312         do 14 i = 1, nlocal
313            id(i) = 1.0d0
314    14   continue
315   c
316         jloc = 0
317         do 15 j = jybegin, jyend
318            yfact = dy*dble(j)
319            offset = jloc*mxsub
320            iloc = 0
321            do 16 i = ixbegin, ixend
322               xfact = dx*dble(i)
323               loc = offset+iloc
324               uu(loc+1) = 16.0d0*xfact*(1.0d0-xfact)*yfact*(1.0d0-yfact)
325               if (i .eq. 0 .or. i .eq. mx-1) then
326                  id(loc+1) = 0.0d0
327               endif
328               if (j .eq. 0 .or. j .eq. my-1) then
329                  id(loc+1) = 0.0d0
330               endif
331               iloc = iloc+1
332    16       continue
333            jloc = jloc+1
334    15   continue
335   c
336         do 17 i = 1, nlocal
337            up(i) = 0.0d0
338            constr(i) = 1.0d0
339    17   continue
340   c
341         call fidaresfun(0.0d0, uu, up, res, ipar, rpar, reserr)
342   c
343         do 18 i = 1, nlocal
344            up(i) = -1.0d0*res(i)
345    18   continue
346   c
347         return
348         end
349   c
350   c ==========
351   c
352         subroutine fidaresfun(tres, u, up, res, ipar, rpar, reserr)
```

```
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub, ipar(*)
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      integer reserr
      double precision tres
      double precision u(*), up(*), res(*)
c
      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
     &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
     &              ixsub, jysub, thispe
c
      call fidacommfn(nlocal, tres, u, up, ipar, rpar, reserr)
c
      call fidaglocfn(nlocal, tres, u, up, res, ipar, rpar, reserr)
c
      return
      end
c
c ==========
c
      subroutine fidacommfn(nloc, tres, u, up, ipar, rpar, reserr)
c
      include "mpif.h"
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub, ipar(*)
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      integer*4 nloc
      integer reserr
      double precision tres, u(*), up(*)
c
      integer request(mpi_status_size)
      double precision buffer(2*mysub)
c
      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
     &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
     &              ixsub, jysub, thispe
c
```

```
412         call brecvpost(request, mxsub, mysub, buffer)
413   c
414         call bsend(mxsub, mysub, u)
415   c
416         call brecvwait(request, mxsub, buffer)
417   c
418         return
419         end
420   c
421   c ==========
422   c
423         subroutine fidaglocfn(nloc, tres, u, up, res, ipar, rpar, reserr)
424   c
425   c global variables
426   c
427         integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
428         integer*4 ixsub, jysub, ipar(*)
429         integer thispe
430         integer mxsubg, mysubg, nlocalg
431         parameter (mxsubg = 5, mysubg = 5)
432         parameter (nlocalg = mxsubg*mysubg)
433         double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
434         double precision uext((mxsubg+2)*(mysubg+2))
435   c
436   c local variables
437   c
438         integer*4 nloc
439         integer reserr
440         double precision tres, u(*), up(*), res(*)
441   c
442         integer*4 i, lx, ly, offsetu, offsetue, locu, locue
443         integer*4 ixbegin, ixend, jybegin, jyend, mxsub2
444         double precision termx, termy, termctr
445   c
446         common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
447        &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
448        &              ixsub, jysub, thispe
449   c
450         mxsub2 = mxsub+2
451   c
452         do 19 i = 1, nlocal
453            res(i) = u(i)
454    19   continue
455   c
456         offsetu = 0
457         offsetue = mxsub2+1
458         do 20 ly = 0, mysub-1
459            do 21 lx = 0, mxsub-1
460               uext(offsetue+lx+1) = u(offsetu+lx+1)
461    21       continue
462            offsetu = offsetu+mxsub
463            offsetue = offsetue+mxsub2
464    20   continue
465   c
466         ixbegin = 0
467         ixend = mxsub-1
468         jybegin = 0
469         jyend = mysub-1
470         if (ixsub .eq. 0) then
```

```
471            ixbegin = ixbegin+1
472         endif
473         if (ixsub .eq. npex-1) then
474            ixend = ixend-1
475         endif
476         if (jysub .eq. 0) then
477            jybegin = jybegin+1
478         endif
479         if (jysub .eq. npey-1) then
480            jyend = jyend-1
481         endif
482   c
483         do 22 ly = jybegin, jyend
484            do 23 lx = ixbegin, ixend
485               locu = lx+ly*mxsub
486               locue = (lx+1)+(ly+1)*mxsub2
487               termx = coeffx*(uext(locue)+uext(locue+2))
488               termy = coeffy*(uext(locue-mxsub2+1)+uext(locue+mxsub2+1))
489               termctr = coeffxy*uext(locue+1)
490               res(locu+1) = up(locu+1)-(termx+termy-termctr)
491    23       continue
492    22    continue
493   c
494         return
495         end
496   c
497   c ==========
498   c
499         subroutine bsend(dsizex, dsizey, uarray)
500   c
501         include "mpif.h"
502   c
503   c global variables
504   c
505         integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
506         integer*4 ixsub, jysub
507         integer thispe
508         integer mxsubg, mysubg, nlocalg
509         parameter (mxsubg = 5, mysubg = 5)
510         parameter (nlocalg = mxsubg*mysubg)
511         double precision dx, dy, coeffx, coeffy, coeffxy
512         double precision uext((mxsubg+2)*(mysubg+2))
513   c
514   c local variables
515   c
516         integer*4 dsizex, dsizey
517         double precision uarray(*)
518   c
519         integer ier
520         double precision bufleft(mysub), bufright(mysub)
521   c
522         common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
523        &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
524        &              ixsub, jysub, thispe
525   c
526         if (jysub .ne. 0) then
527            call mpi_send(uarray(1), dsizex, mpi_double_precision,
528        &                 thispe-npex, 0, mpi_comm_world, ier)
529         endif
```

```
c
      if (jysub .ne. npey-1) then
         offsetu = (mysub-1)*dsizex
         call mpi_send(uarray(offsetu+1), dsizex, mpi_double_precision,
     &                 thispe+npex, 0, mpi_comm_world, ier)
      endif
c
      if (ixsub .ne. 0) then
         do 24 ly = 0, mysub-1
            offsetu = ly*dsizex
            bufleft(ly+1) = uarray(offsetu+1)
 24      continue
         call mpi_send(bufleft(1), dsizey, mpi_double_precision,
     &                 thispe-1, 0, mpi_comm_world, ier)
      endif
c
      if (ixsub .ne. npex-1) then
         do 25 ly = 0, mysub-1
            offsetu = ly*mxsub+(mxsub-1)
            bufright(ly+1) = uarray(offsetu+1)
 25      continue
         call mpi_send(bufright(1), dsizey, mpi_double_precision,
     &                 thispe+1, 0, mpi_comm_world, ier)
      endif
c
      return
      end
c
c ==========
c
      subroutine brecvpost(request, dsizex, dsizey, buffer)
c
      include "mpif.h"
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      integer*4 dsizex, dsizey
      integer request(*)
      double precision buffer(*)
c
      integer ier
      integer*4 offsetue
c
      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
     &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
     &              ixsub, jysub, thispe
c
      if (jysub .ne. 0) then
```

```
589           call mpi_irecv(uext(2), dsizex, mpi_double_precision,
590      &                   thispe-npex, 0, mpi_comm_world, request(1),
591      &                   ier)
592           endif
593   c
594           if (jysub .ne. npey-1) then
595              offsetue = (1+(mysub+1)*(mxsub+2))
596              call mpi_irecv(uext(offsetue+1), dsizex, mpi_double_precision,
597      &                   thispe+npex, 0, mpi_comm_world, request(2),
598      &                   ier)
599           endif
600   c
601           if (ixsub .ne. 0) then
602              call mpi_irecv(buffer(1), dsizey, mpi_double_precision,
603      &                   thispe-1, 0, mpi_comm_world, request(3),
604      &                   ier)
605           endif
606   c
607           if (ixsub .ne. npex-1) then
608              call mpi_irecv(buffer(1+mysub), dsizey, mpi_double_precision,
609      &                   thispe+1, 0, mpi_comm_world, request(4),
610      &                   ier)
611           endif
612   c
613           return
614           end
615   c
616   c ==========
617   c
618           subroutine brecvwait(request, dsizex, buffer)
619   c
620           include "mpif.h"
621   c
622   c global variables
623   c
624           integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
625           integer*4 ixsub, jysub
626           integer thispe
627           integer mxsubg, mysubg, nlocalg
628           parameter (mxsubg = 5, mysubg = 5)
629           parameter (nlocalg = mxsubg*mysubg)
630           double precision dx, dy, coeffx, coeffy, coeffxy
631           double precision uext((mxsubg+2)*(mysubg+2))
632   c
633   c local variables
634   c
635           integer request(*)
636           integer*4 dsizex
637           double precision buffer(*)
638   c
639           integer*4 ly, dsizex2, offsetue
640           integer ier, status(mpi_status_size)
641   c
642           common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
643      &                  nlocal, neq, mx, my, mxsub, mysub, npey, npex,
644      &                  ixsub, jysub, thispe
645   c
646           dsizex2 = dsizex+2
647   c
```

```
      if (jysub .ne. 0) then
         call mpi_wait(request(1), status, ier)
      endif
c
      if (jysub .ne. npey-1) then
         call mpi_wait(request(2), status, ier)
      endif
c
      if (ixsub .ne. 0) then
         call mpi_wait(request(3), status, ier)
         do 26 ly = 0, mysub-1
            offsetue = (ly+1)*dsizex2
            uext(offsetue+1) = buffer(ly+1)
 26      continue
      endif
c
      if (ixsub .ne. npex-1) then
         call mpi_wait(request(4), status, ier)
         do 27 ly = 0, mysub-1
            offsetue = (ly+2)*dsizex2-1
            uext(offsetue+1) = buffer(ly+mysub+1)
 27      continue
      endif
c
      return
      end
c
c ==========
c
      subroutine prntoutput(tret, u, iout, rout)
c
c global variables
c
      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
      integer*4 ixsub, jysub
      integer thispe
      integer mxsubg, mysubg, nlocalg
      parameter (mxsubg = 5, mysubg = 5)
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy
      double precision uext((mxsubg+2)*(mysubg+2))
c
c  local variables
c
      integer*4 iout(*), lenrwbbd, leniwbbd, ngebbd
      double precision tret, umax, u(*), rout(*)
c
      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
     &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
     &              ixsub, jysub, thispe
c
      call maxnorm(u, umax)
c
      if (thispe .eq. 0) then
         call fidabbdopt(lenrwbbd, leniwbbd, ngebbd)
         write(*,28) tret, umax, iout(9), iout(3), iout(7),
     &               iout(20), iout(4), iout(16), ngebbd, rout(2),
     &               iout(18), iout(19)
 28      format(' ', e10.4, ' ', e13.5, '  ', i1, '  ', i2,
```

```
707       &           '  ', i3, '  ', i3, '  ', i2,'+',i2, '  ',
708       &           i3, '  ', e9.2, '  ', i2, '  ', i3)
709        endif
710   c
711        return
712        end
713   c
714   c ==========
715   c
716        subroutine maxnorm(u, unorm)
717   c
718        include "mpif.h"
719   c
720   c global variables
721   c
722        integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
723        integer*4 ixsub, jysub
724        integer thispe
725        integer mxsubg, mysubg, nlocalg
726        parameter (mxsubg = 5, mysubg = 5)
727        parameter (nlocalg = mxsubg*mysubg)
728        double precision dx, dy, coeffx, coeffy, coeffxy
729        double precision uext((mxsubg+2)*(mysubg+2))
730   c
731   c local variables
732   c
733        integer*4 i
734        integer ier
735        double precision temp, unorm, u(*)
736   c
737        common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
738       &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
739       &              ixsub, jysub, thispe
740   c
741        temp = 0.0d0
742   c
743        do 29 i = 1, nlocal
744           temp = max(abs(u(i)), temp)
745    29    continue
746   c
747        call mpi_allreduce(temp, unorm, 1, mpi_double_precision,
748       &                   mpi_max, mpi_comm_world, ier)
749   c
750   c      unorm = temp
751   c
752        return
753        end
754   c
755   c ==========
756   c
757        subroutine prntintro(rtol, atol)
758   c
759   c global variables
760   c
761        integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
762        integer*4 ixsub, jysub
763        integer thispe
764        integer mxsubg, mysubg, nlocalg
765        parameter (mxsubg = 5, mysubg = 5)
```

```
      parameter (nlocalg = mxsubg*mysubg)
      double precision dx, dy, coeffx, coeffy, coeffxy
      double precision uext((mxsubg+2)*(mysubg+2))
c
c local variables
c
      double precision rtol, atol
c
      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
     &              nlocal, neq, mx, my, mxsub, mysub, npey, npex,
     &              ixsub, jysub, thispe
c
      write(*,30) mx, my, neq, mxsub, mysub, npex, npey, rtol, atol
 30   format(/'fidakryx_bbd_p: Heat equation, parallel example problem',
     &     ' for FIDA', /, 16x,'Discretized heat equation',
     &     ' on 2D unit square.', /, 16x,'Zero boundary',
     &     ' conditions, polynomial conditions.', /,
     &     16x,'Mesh dimensions: ', i2, ' x ', i2,
     &     '          Total system size: ', i3, //,
     &     'Subgrid dimensions: ', i2, ' x ', i2,
     &     '           Processor array: ', i2, ' x ', i2, /,
     &     'Tolerance parameters: rtol = ', e8.2, '   atol = ',
     &     e8.2, /, 'Constraints set to force all solution',
     &     ' components >= 0.', /, 'SUPPRESSALG = TRUE to remove',
     &     ' boundary components from the error test.', /,
     &     'Linear solver: SPGMR.    Preconditioner: BBDPRE - ',
     &     'Banded-block-diagonal.')
c
      return
      end
c
c ==========
c
      subroutine prntcase(num, mudq, mukeep)
c
c local variables
c
      integer*4 mudq, mukeep
      integer num
c
      write(*,31) num, mudq, mukeep
 31   format(//, 'Case ', i2, /, '  Difference quotient half-',
     &     'bandwidths =', i2, /, '  Retained matrix half-bandwidths =',
     &     i2, //, 'Output Summary',/,'  umax = max-norm of solution',
     &     /,'  nre = nre + nreLS (total number of RES evals.)',
     &     //, '   time         umax       k nst  nni  nli   nre',
     &     '   nge      h     npe  nps', /,
     &     '------------------------------------------------------',
     &     '------------------')
c
      return
      end
c
c ==========
c
      subroutine prntfinalstats(iout)
c
c local variables
c
```

```
      integer*4 iout(*)
c
      write(*,32) iout(5), iout(6), iout(21)
 32   format(/, 'Error test failures            =', i3, /,
     &     'Nonlinear convergence failures =', i3, /,
     &     'Linear convergence failures    =', i3)
c
      return
      end
```