Example Programs for IDA v2.2.1

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

January 2005

UCRL-SM-208113

DISCLAIMER

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

Contents

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

- `irobx` solves the Robertson chemical kinetics problem [3] which consists of two differential equations and one algebraic constraint.

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

- `iheatsb` 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.

- `iheatsk` solves the same problem as `iheatsb`, with the Krylov linear solver IDASPGMR. The preconditioner uses only the diagonal elements of the Jacobian.

- `iwebsb` 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.

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

- `iheatpk` solves the same problem as `iheatbbd` with a user-supplied preconditioner which uses the diagonal elements of the Jacobian only.

- `iheatbbd` is a parallel implementation of the 2-D heat equation.

  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.

- `iwebpk` solves the same problem as `iwebbbd` with a user-supplied preconditioner.

  The precondtioner 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).

- `iwebbbd` is a parallel implementation of the predator-prey problem.

  The problem is solved in parallel using the IDASPGMR linear solver and the IDABBDPRE preconditioner.

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. §5.1) 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.

2 Serial example problems

2.1 A dense example: `irobx`

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 $t = 4 \cdot 10^{10}$.

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 `sundialstypes.h` file provides the definition of the type `realtype` (see §5.1 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 `idadense.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 `idadense.h` also includes the `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 §5.5.2.

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 §5.4.1 for full details of this call.

The calls to IDADense (see §5.4.2) and IDADenseSetJacFn (see §5.4.5) 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 t = tout. If an error occured during the call to IDASolve, the program returns 1 and terminates. On a successful return (indicated by a return value IDA_SUCCESS), the program prints the solution, the cumulative number of steps taken so far, and the current method order and step size.

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 NV_Destroy_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 (nreD), the number of Jacobian evaluations (njeD), 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 §5.4.7.

The functions resrob (of type IDAResFn) and jacrob (of type IDADenseJacFn) are straightforward expressions of the DAE system. The function jacrob makes use of the macro IJth discussed above. See §5.5.1 for detailed specifications of IDAResFn.

Sample output from irobx follows.

irobx sample output

```
irobx: 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-06 1e-10 1e-06
Initial conditions y0 = (1 0 0)
Constraints and id not used.

------------------------------------------------------------------------
  t           y1           y2           y3      | nst  k      h
------------------------------------------------------------------------
4.00e-01   9.8517e-01   3.3864e-05   1.4795e-02 |  77  3   1.1431e-01
```

```
4.00e+00   9.0551e-01   2.2403e-05   9.4470e-02 |  91  4   3.7037e-01
4.00e+01   7.1582e-01   9.1854e-06   2.8417e-01 | 127  4   2.9630e+00
4.00e+02   4.5051e-01   3.2227e-06   5.4949e-01 | 177  3   1.2405e+01
4.00e+03   1.8316e-01   8.9396e-07   8.1684e-01 | 228  3   2.7646e+02
4.00e+04   3.8985e-02   1.6218e-07   9.6101e-01 | 278  5   2.6140e+03
4.00e+05   4.9388e-03   1.9852e-08   9.9506e-01 | 324  5   2.7701e+04
4.00e+06   5.1763e-04   2.0716e-09   9.9948e-01 | 355  4   3.9788e+05
4.00e+07   5.1907e-05   2.0764e-10   9.9995e-01 | 380  3   6.3661e+06
4.00e+08   5.8818e-06   2.3527e-11   9.9999e-01 | 394  1   9.1671e+07
4.00e+09   7.0539e-07   2.8216e-12   1.0000e-00 | 402  1   1.4667e+09
4.00e+10  -7.3001e-07  -2.9200e-12   1.0000e+00 | 407  1   2.3468e+10

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
```

2.2 A banded example: `iwebsb`

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 `iwebsb.c` program (listed in Appendix B) includes the file `idaband.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 §8.2. 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 `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 `irobx`. 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 `NV_Destroy_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 (`nreB`), the number of Jacobian evaluations (`njeB`), 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 §5.4.7.

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 `iwebsb` follows.

iwebsb sample output

```
iwebsb: 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: `iheatsk`

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, `idaspgmr.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 `iheatsk` follows.

iheatsk sample output

```
iheatsk: 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   nreS     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   5.54028e-21  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   nreS     h      npe nps
------------------------------------------------------------------------
  0.00   9.75461e-01  0    0    0   77     0    77   0.00e+00   18 134
  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   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 Parallel example problems

3.1 A user preconditioner example: iheatpk

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 `iheatpk` which solves the same 2-D heat PDE problem as `iheatsk`. 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 `iheatpk` main program begins with MPI calls to initialize MPI and to set multi-processor 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 `iheatsk`, 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, `iheatsk` 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 `iheatpk` follows.

iheatpk sample output

```
iheatpk: 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   nreS     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   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
  1.28   3.85107e-20  1   41   53   77    53    77   4.21e-01   14 130
  2.56   5.00523e-20  1   43   55   77    55    77   1.69e+00   16 132
  5.12   1.58940e-19  1   44   56   77    56    77   3.37e+00   17 133
 10.24   5.12685e-19  1   45   57   77    57    77   6.74e+00   18 134

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

3.2 An IDABBDPRE preconditioner example: `iwebbbd`

In this example, `iwebbbd`, we solve the same food web problem as with `iwebsb`, 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 `iheatpk`, 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 `iwebbbd` 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 `iwebbbd` follows.

iwebbbd sample output

```
iwebbbd: Predator-prey DAE parallel example problem for IDA

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:  rtol = 1e-05   atol = 1e-05
Linear solver: IDASPGMR     Max. Krylov dimension maxl: 12
Preconditioner: band-block-diagonal (IDABBDPRE), with parameters
     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   | 111  5   1.6153e-04
           1.6189e+06   1.9735e+06   |

1.00e-01   2.4019e+02   2.7072e+02   | 191  1   4.1353e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+02   | 194  1   3.3082e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+02   | 194  1   3.3082e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+02   | 195  1   6.6164e-01
           2.4019e+06   2.7072e+06   |

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

Final statistics:

Number of steps                    = 195
Number of residual evaluations     = 895
Number of nonlinear iterations     = 239
Number of error test failures      = 0
Number of nonlinear conv. failures = 0
```

```
Number of linear iterations        = 654
Number of linear conv. failures    = 0

Number of preconditioner setups    = 26
Number of preconditioner solves    = 895
Number of local residual evals.    = 1092
```

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 v2.2.0. Technical Report UCRL-SM-208112, LLNL, 2004.

[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 `irobx.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.16.2.2 $
 * $Date: 2005/03/17 22:50:54 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * This simple example problem for IDA/IDAS, 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.
 *
 * The problem is solved with IDA/IDAS 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 "sundialstypes.h"
#include "sundialsmath.h"
#include "nvector_serial.h"
#include "ida.h"
#include "idadense.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);

int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
           N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
```

```
53            N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);
54
55  /* Prototypes of private functions */
56  static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y);
57  static void PrintOutput(void *mem, realtype t, N_Vector y);
58  static void PrintFinalStats(void *mem);
59  static int check_flag(void *flagvalue, char *funcname, int opt);
60
61  /*
62   *-------------------------------------------------------------------
63   * MAIN PROGRAM
64   *-------------------------------------------------------------------
65   */
66
67  int main(void)
68  {
69    void *mem;
70    N_Vector yy, yp, avtol;
71    realtype rtol, *yval, *ypval, *atval;
72    realtype t0, t1, tout, tret;
73    int iout, retval;
74
75    mem = NULL;
76    yy = yp = avtol = NULL;
77    yval = ypval = atval = NULL;
78
79    /* Allocate N-vectors. */
80
81    yy = N_VNew_Serial(NEQ);
82    if(check_flag((void *)yy, "N_VNew_Serial", 0)) return(1);
83    yp = N_VNew_Serial(NEQ);
84    if(check_flag((void *)yp, "N_VNew_Serial", 0)) return(1);
85    avtol = N_VNew_Serial(NEQ);
86    if(check_flag((void *)avtol, "N_VNew_Serial", 0)) return(1);
87
88    /* Create and initialize  y, y', and absolute tolerance vectors. */
89
90    yval  = NV_DATA_S(yy);
91    yval[0] = ONE;
92    yval[1] = ZERO;
93    yval[2] = ZERO;
94
95    ypval = NV_DATA_S(yp);
96    ypval[0]  = RCONST(-0.04);
97    ypval[1]  = RCONST(0.04);
98    ypval[2]  = ZERO;
99
100   rtol = RCONST(1.0e-4);
101
102   atval = NV_DATA_S(avtol);
103   atval[0] = RCONST(1.0e-6);
104   atval[1] = RCONST(1.0e-10);
105   atval[2] = RCONST(1.0e-6);
106
```

```
107   /* Integration limits */
108
109   t0 = ZERO;
110   t1 = RCONST(0.4);
111
112   /* Call IDACreate and IDAMalloc to initialize IDA memory */
113
114   mem = IDACreate();
115   if(check_flag((void *)mem, "IDACreate", 0)) return(1);
116   retval = IDAMalloc(mem, resrob, t0, yy, yp, IDA_SV, &rtol, avtol);
117   if(check_flag(&retval, "IDAMalloc", 1)) return(1);
118
119   /* Call IDADense and set up the linear solver. */
120
121   retval = IDADense(mem, NEQ);
122   if(check_flag(&retval, "IDADense", 1)) return(1);
123   retval = IDADenseSetJacFn(mem, jacrob);
124   if(check_flag(&retval, "IDADenseSetJacFn", 1)) return(1);
125
126   /* Loop over tout values and call IDASolve. */
127
128   PrintHeader(rtol, avtol, yy);
129
130   for (tout = t1, iout = 1; iout <= NOUT ; iout++, tout *= RCONST(10.0)) {
131     retval=IDASolve(mem, tout, &tret, yy, yp, IDA_NORMAL);
132     if(check_flag(&retval, "IDASolve", 1)) return(1);
133     PrintOutput(mem,tret,yy);
134   }
135
136   PrintFinalStats(mem);
137
138   /* Free memory */
139
140   IDAFree(mem);
141   N_VDestroy_Serial(yy);
142   N_VDestroy_Serial(yp);
143   N_VDestroy_Serial(avtol);
144
145   return(0);
146
147 }
148
149 /*
150  *--------------------------------------------------------------------
151  * FUNCTIONS CALLED BY IDA
152  *--------------------------------------------------------------------
153  */
154
155 /*
156  * Define the system residual function.
157  */
158
159 int resrob(realtype tres, N_Vector yy, N_Vector yp, N_Vector rr, void *rdata)
160 {
```

```
161   realtype *yval, *ypval, *rval;
162
163   yval = NV_DATA_S(yy);
164   ypval = NV_DATA_S(yp);
165   rval = NV_DATA_S(rr);
166
167   rval[0]  = RCONST(-0.04)*yval[0] + RCONST(1.0e4)*yval[1]*yval[2];
168   rval[1]  = -rval[0] - RCONST(3.0e7)*yval[1]*yval[1] - ypval[1];
169   rval[0] -=  ypval[0];
170   rval[2]  =  yval[0] + yval[1] + yval[2] - ONE;
171
172   return(0);
173
174 }
175
176 /*
177  * Define the Jacobian function.
178  */
179
180 int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
181            N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
182            N_Vector tempv1, N_Vector tempv2, N_Vector tempv3)
183 {
184
185   realtype *yval;
186
187   yval = NV_DATA_S(yy);
188
189   IJth(JJ,1,1) = RCONST(-0.04) - cj;
190   IJth(JJ,2,1) = RCONST(0.04);
191   IJth(JJ,3,1) = ONE;
192   IJth(JJ,1,2) = RCONST(1.0e4)*yval[2];
193   IJth(JJ,2,2) = RCONST(-1.0e4)*yval[2] - RCONST(6.0e7)*yval[1] - cj;
194   IJth(JJ,3,2) = ONE;
195   IJth(JJ,1,3) = RCONST(1.0e4)*yval[1];
196   IJth(JJ,2,3) = RCONST(-1.0e4)*yval[1];
197   IJth(JJ,3,3) = ONE;
198
199   return(0);
200
201 }
202
203 /*
204  *-------------------------------------------------------------------
205  * PRIVATE FUNCTIONS
206  *-------------------------------------------------------------------
207  */
208
209 /*
210  * Print first lines of output (problem description)
211  */
212
213 static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y)
214 {
```

```
  realtype *atval, *yval;

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

  printf("\nirobx: 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("%8.2Le %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("%8.2le %12.4le %12.4le %12.4le | %3ld  %1d %12.4le\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#else
  printf("%8.2e %12.4e %12.4e %12.4e | %3ld  %1d %12.4e\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#endif
}

/*
 * Print final integrator statistics
 */

static void PrintFinalStats(void *mem)
{
  int retval;
  long int nst, nni, njeD, nre, nreD, netf, ncfn;

  retval = IDAGetNumSteps(mem, &nst);
  check_flag(&retval, "IDAGetNumSteps", 1);
  retval = IDAGetNumResEvals(mem, &nre);
  check_flag(&retval, "IDAGetNumResEvals", 1);
  retval = IDADenseGetNumJacEvals(mem, &njeD);
  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, &nreD);
  check_flag(&retval, "IDADenseGetNumResEvals", 1);

  printf("\nFinal Run Statistics: \n\n");
  printf("Number of steps                    = %ld\n", nst);
  printf("Number of residual evaluations     = %ld\n", nre+nreD);
  printf("Number of Jacobian evaluations     = %ld\n", njeD);
  printf("Number of nonlinear iterations     = %ld\n", nni);
  printf("Number of error test failures      = %ld\n", netf);
  printf("Number of nonlinear conv. failures = %ld\n", ncfn);

}

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

  return(0);
}
```

B Listing of `iwebsb.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.18.2.1 $
 * $Date: 2005/03/17 22:50:54 $
 * -----------------------------------------------------------------
 * 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 "sundialstypes.h"   /* Definitions of realtype and booleantype       */
#include "ida.h"             /* Main header file                              */
#include "idaband.h"         /* Use IDABAND linear solver                     */
#include "nvector_serial.h"  /* Definitions of type N_Vector, macro NV_DATA_S */
#include "smalldense.h"      /* Contains definitions for denalloc routine     */

/* 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);
```

```
161 static int check_flag(void *flagvalue, char *funcname, int opt);
162
163 /*
164  *--------------------------------------------------------------------
165  * MAIN PROGRAM
166  *--------------------------------------------------------------------
167  */
168
169 int main()
170 {
171   void *mem;
172   UserData webdata;
173   N_Vector cc, cp, id;
174   int iout, retval;
175   long int mu, ml;
176   realtype rtol, atol, t0, tout, tret;
177
178   mem = NULL;
179   webdata = NULL;
180   cc = cp = id = NULL;
181
182   /* Allocate and initialize user data block webdata. */
183
184   webdata = (UserData) malloc(sizeof *webdata);
185   webdata->rates = N_VNew_Serial(NEQ);
186   webdata->acoef = denalloc(NUM_SPECIES);
187
188   InitUserData(webdata);
189
190   /* Allocate N-vectors and initialize cc, cp, and id. */
191
192   cc  = N_VNew_Serial(NEQ);
193   if(check_flag((void *)cc, "N_VNew_Serial", 0)) return(1);
194
195   cp  = N_VNew_Serial(NEQ);
196   if(check_flag((void *)cp, "N_VNew_Serial", 0)) return(1);
197
198   id  = N_VNew_Serial(NEQ);
199   if(check_flag((void *)id, "N_VNew_Serial", 0)) return(1);
200
201   SetInitialProfiles(cc, cp, id, webdata);
202
203   /* Set remaining inputs to IDAMalloc. */
204
205   t0 = ZERO;
206   rtol = RTOL;
207   atol = ATOL;
208
209   /* Call IDACreate and IDAMalloc to initialize IDA. */
210
211   mem = IDACreate();
212   if(check_flag((void *)mem, "IDACreate", 0)) return(1);
213
214   retval = IDASetRdata(mem, webdata);
```

```
215   if(check_flag(&retval, "IDASetRdata", 1)) return(1);
216
217   retval = IDASetId(mem, id);
218   if(check_flag(&retval, "IDASetId", 1)) return(1);
219
220   retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, &rtol, &atol);
221   if(check_flag(&retval, "IDAMalloc", 1)) return(1);
222
223   /* Call IDABand to specify the IDA linear solver. */
224
225   mu = ml = NSMX;
226   retval = IDABand(mem, NEQ, mu, ml);
227   if(check_flag(&retval, "IDABand", 1)) return(1);
228
229   /* Call IDACalcIC (with default options) to correct the initial values. */
230
231   tout = RCONST(0.001);
232   retval = IDACalcIC(mem, IDA_YA_YDP_INIT, tout);
233   if(check_flag(&retval, "IDACalcIC", 1)) return(1);
234
235   /* Print heading, basic parameters, and initial values. */
236
237   PrintHeader(mu, ml, rtol, atol);
238   PrintOutput(mem, cc, ZERO);
239
240   /* Loop over iout, call IDASolve (normal mode), print selected output. */
241
242   for (iout = 1; iout <= NOUT; iout++) {
243
244     retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
245     if(check_flag(&retval, "IDASolve", 1)) return(retval);
246
247     PrintOutput(mem, cc, tret);
248
249     if (iout < 3) tout *= TMULT; else tout += TADD;
250
251   }
252
253   /* Print final statistics and free memory. */
254
255   PrintFinalStats(mem);
256
257   /* Free memory */
258
259   IDAFree(mem);
260
261   N_VDestroy_Serial(cc);
262   N_VDestroy_Serial(cp);
263   N_VDestroy_Serial(id);
264
265
266   denfree(webdata->acoef);
267   N_VDestroy_Serial(webdata->rates);
268   free(webdata);
```

```
269
270   return(0);
271 }
272
273 /* Define lines for readability in later routines */
274
275 #define acoef  (webdata->acoef)
276 #define bcoef  (webdata->bcoef)
277 #define cox    (webdata->cox)
278 #define coy    (webdata->coy)
279
280 /*
281  *--------------------------------------------------------------------
282  * FUNCTIONS CALLED BY IDA
283  *--------------------------------------------------------------------
284  */
285
286 /*
287  * resweb: System residual function for predator-prey system.
288  * This routine calls Fweb to get all the right-hand sides of the
289  * equations, then loads the residual vector accordingly,
290  * using cp in the case of prey species.
291  */
292
293 static int resweb(realtype tt, N_Vector cc, N_Vector cp,
294                   N_Vector res,  void *rdata)
295 {
296   long int jx, jy, is, yloc, loc, np;
297   realtype *resv, *cpv;
298   UserData webdata;
299
300   webdata = (UserData)rdata;
301
302   cpv = NV_DATA_S(cp);
303   resv = NV_DATA_S(res);
304   np = webdata->np;
305
306   /* Call Fweb to set res to vector of right-hand sides. */
307   Fweb(tt, cc, res, webdata);
308
309   /* Loop over all grid points, setting residual values appropriately
310      for differential or algebraic components.                        */
311
312   for (jy = 0; jy < MY; jy++) {
313     yloc = NSMX * jy;
314     for (jx = 0; jx < MX; jx++) {
315       loc = yloc + NUM_SPECIES * jx;
316       for (is = 0; is < NUM_SPECIES; is++) {
317         if (is < np)
318           resv[loc+is] = cpv[loc+is] - resv[loc+is];
319         else
320           resv[loc+is] = -resv[loc+is];
321       }
322     }
```

```
  }

  return(0);

}

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

/*
 * InitUserData: Load problem constants in webdata (of type UserData).
 */

static void InitUserData(UserData webdata)
{
  int i, j, np;
  realtype *a1,*a2, *a3, *a4, dx2, dy2;

  webdata->mx = MX;
  webdata->my = MY;
  webdata->ns = NUM_SPECIES;
  webdata->np = NPREY;
  webdata->dx = AX/(MX-1);
  webdata->dy = AY/(MY-1);
  webdata->Neq= NEQ;

  /* Set up the coefficients a and b, and others found in the equations. */
  np = webdata->np;
  dx2 = (webdata->dx)*(webdata->dx); dy2 = (webdata->dy)*(webdata->dy);

  for (i = 0; i < np; i++) {
    a1 = &(acoef[i][np]);
    a2 = &(acoef[i+np][0]);
    a3 = &(acoef[i][0]);
    a4 = &(acoef[i+np][np]);
    /*  Fill in the portion of acoef in the four quadrants, row by row. */
    for (j = 0; j < np; j++) {
      *a1++ =  -GG;
      *a2++ =   EE;
      *a3++ = ZERO;
      *a4++ = ZERO;
    }

    /* Reset the diagonal elements of acoef to -AA. */
    acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;

    /* Set coefficients for b and diffusion terms. */
    bcoef[i] = BB; bcoef[i+np] = -BB;
    cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
    coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
  }
```

```
377
378 }
379
380 /*
381  * SetInitialProfiles: Set initial conditions in cc, cp, and id.
382  * A polynomial profile is used for the prey cc values, and a constant
383  * (1.0e5) is loaded as the initial guess for the predator cc values.
384  * The id values are set to 1 for the prey and 0 for the predators.
385  * The prey cp values are set according to the given system, and
386  * the predator cp values are set to zero.
387  */
388
389 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
390                                UserData webdata)
391 {
392   long int loc, yloc, is, jx, jy, np;
393   realtype xx, yy, xyfactor, fac;
394   realtype *ccv, *cpv, *idv;
395
396   ccv = NV_DATA_S(cc);
397   cpv = NV_DATA_S(cp);
398   idv = NV_DATA_S(id);
399   np = webdata->np;
400
401   /* Loop over grid, load cc values and id values. */
402   for (jy = 0; jy < MY; jy++) {
403     yy = jy * webdata->dy;
404     yloc = NSMX * jy;
405     for (jx = 0; jx < MX; jx++) {
406       xx = jx * webdata->dx;
407       xyfactor = RCONST(16.0)*xx*(ONE-xx)*yy*(ONE-yy);
408       xyfactor *= xyfactor;
409       loc = yloc + NUM_SPECIES*jx;
410       fac = ONE + ALPHA * xx * yy + BETA * sin(FOURPI*xx) * sin(FOURPI*yy);
411
412       for (is = 0; is < NUM_SPECIES; is++) {
413         if (is < np) {
414             ccv[loc+is] = RCONST(10.0) + (realtype)(is+1) * xyfactor;
415           idv[loc+is] = ONE;
416         }
417         else {
418           ccv[loc+is] = RCONST(1.0e5);
419           idv[loc+is] = ZERO;
420         }
421       }
422     }
423   }
424
425   /* Set c' for the prey by calling the function Fweb. */
426   Fweb(ZERO, cc, cp, webdata);
427
428   /* Set c' for predators to 0. */
429   for (jy = 0; jy < MY; jy++) {
430     yloc = NSMX * jy;
```

```
    for (jx = 0; jx < MX; jx++) {
      loc = yloc + NUM_SPECIES * jx;
      for (is = np; is < NUM_SPECIES; is++) {
        cpv[loc+is] = ZERO;
      }
    }
  }
}

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

static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol)
{
  printf("\niwebsb: Predator-prey DAE serial example problem for IDA \n\n");
  printf("Number of species ns: %d", NUM_SPECIES);
  printf("     Mesh dimensions: %d x %d", MX, MY);
  printf("     System size: %d\n", NEQ);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Linear solver: IDABAND,  Band parameters mu = %ld, ml = %ld\n",mu,ml);
  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 the concentrations
 * 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);

```

```
485   c_bl = IJ_Vptr(c,0,0);
486   c_tr = IJ_Vptr(c,MX-1,MY-1);
487 
488 #if defined(SUNDIALS_EXTENDED_PRECISION)
489   printf("%8.2Le %12.4Le %12.4Le   | %3ld  %1d %12.4Le\n",
490          t, c_bl[0], c_tr[1], nst, kused, hused);
491   for (i=1;i<NUM_SPECIES;i++)
492     printf("         %12.4Le %12.4Le   |\n",c_bl[i],c_tr[i]);
493 #elif defined(SUNDIALS_DOUBLE_PRECISION)
494   printf("%8.2le %12.4le %12.4le   | %3ld  %1d %12.4le\n",
495          t, c_bl[0], c_tr[1], nst, kused, hused);
496   for (i=1;i<NUM_SPECIES;i++)
497     printf("         %12.4le %12.4le   |\n",c_bl[i],c_tr[i]);
498 #else
499   printf("%8.2e %12.4e %12.4e   | %3ld  %1d %12.4e\n",
500          t, c_bl[0], c_tr[1], nst, kused, hused);
501   for (i=1;i<NUM_SPECIES;i++)
502     printf("         %12.4e %12.4e   |\n",c_bl[i],c_tr[i]);
503 #endif
504 
505   printf("\n");
506 }
507 
508 /*
509  * PrintFinalStats: Print final run data contained in iopt.
510  */
511 
512 static void PrintFinalStats(void *mem)
513 {
514   long int nst, nre, nreB, nni, nje, netf, ncfn;
515   int flag;
516 
517   flag = IDAGetNumSteps(mem, &nst);
518   check_flag(&flag, "IDAGetNumSteps", 1);
519   flag = IDAGetNumNonlinSolvIters(mem, &nni);
520   check_flag(&flag, "IDAGetNumNonlinSolvIters", 1);
521   flag = IDAGetNumResEvals(mem, &nre);
522   check_flag(&flag, "IDAGetNumResEvals", 1);
523   flag = IDAGetNumErrTestFails(mem, &netf);
524   check_flag(&flag, "IDAGetNumErrTestFails", 1);
525   flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
526   check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1);
527   flag = IDABandGetNumJacEvals(mem, &nje);
528   check_flag(&flag, "IDABandGetNumJacEvals", 1);
529   flag = IDABandGetNumResEvals(mem, &nreB);
530   check_flag(&flag, "IDABandGetNumResEvals", 1);
531 
532   printf("-------------------------------------------------------------\n");
533   printf("Final run statistics: \n\n");
534   printf("Number of steps                    = %ld\n", nst);
535   printf("Number of residual evaluations     = %ld\n", nre+nreB);
536   printf("Number of Jacobian evaluations     = %ld\n", nje);
537   printf("Number of nonlinear iterations     = %ld\n", nni);
538   printf("Number of error test failures      = %ld\n", netf);
```

```
539   printf("Number of nonlinear conv. failures = %ld\n", ncfn);
540 
541 }
542 
543 /*
544  * Fweb: Rate function for the food-web problem.
545  * This routine computes the right-hand sides of the system equations,
546  * consisting of the diffusion term and interaction term.
547  * The interaction term is computed by the function WebRates.
548  */
549 
550 static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate,
551                  UserData webdata)
552 {
553   long int jx, jy, is, idyu, idyl, idxu, idxl;
554   realtype xx, yy, *cxy, *ratesxy, *cratexy, dcyli, dcyui, dcxli, dcxui;
555 
556   /* Loop over grid points, evaluate interaction vector (length ns),
557      form diffusion difference terms, and load crate.                    */
558 
559   for (jy = 0; jy < MY; jy++) {
560     yy = (webdata->dy) * jy ;
561     idyu = (jy!=MY-1) ? NSMX : -NSMX;
562     idyl = (jy!= 0  ) ? NSMX : -NSMX;
563 
564     for (jx = 0; jx < MX; jx++) {
565       xx = (webdata->dx) * jx;
566       idxu = (jx!= MX-1) ?  NUM_SPECIES : -NUM_SPECIES;
567       idxl = (jx!=  0  ) ?  NUM_SPECIES : -NUM_SPECIES;
568       cxy = IJ_Vptr(cc,jx,jy);
569       ratesxy = IJ_Vptr(webdata->rates,jx,jy);
570       cratexy = IJ_Vptr(crate,jx,jy);
571 
572       /* Get interaction vector at this grid point. */
573       WebRates(xx, yy, cxy, ratesxy, webdata);
574 
575       /* Loop over species, do differencing, load crate segment. */
576       for (is = 0; is < NUM_SPECIES; is++) {
577 
578         /* Differencing in y. */
579         dcyli = *(cxy+is) - *(cxy - idyl + is) ;
580         dcyui = *(cxy + idyu + is) - *(cxy+is);
581 
582         /* Differencing in x. */
583         dcxli = *(cxy+is) - *(cxy - idxl + is);
584         dcxui = *(cxy + idxu +is) - *(cxy+is);
585 
586         /* Compute the crate values at (xx,yy). */
587         cratexy[is] = coy[is] * (dcyui - dcyli) +
588           cox[is] * (dcxui - dcxli) + ratesxy[is];
589 
590       } /* End is loop */
591     } /* End of jx loop */
592   } /* End of jy loop */
```

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

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

C Listing of `iheatsk.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.16.2.2 $
 * $Date: 2005/03/17 22:50:54 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for IDA/IDAS: 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 "sundialstypes.h"
#include "nvector_serial.h"
#include "ida.h"
#include "idaspgmr.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 */
```

```
53
54 typedef struct {
55   long int mm;  /* number of grid points */
56   realtype dx;
57   realtype coeff;
58   N_Vector pp;  /* vector of prec. diag. elements */
59 } *UserData;
60
61 /* Prototypes for functions called by IDA */
62
63 int resHeat(realtype tres, N_Vector uu, N_Vector up,
64             N_Vector resval, void *rdata);
65
66 int PsetupHeat(realtype tt,
67                N_Vector uu, N_Vector up, N_Vector rr,
68                realtype c_j, void *prec_data,
69                N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
70
71 int PsolveHeat(realtype tt,
72                N_Vector uu, N_Vector up, N_Vector rr,
73                N_Vector rvec, N_Vector zvec,
74                realtype c_j, realtype delta, void *prec_data,
75                N_Vector tmp);
76
77 /* Prototypes for private functions */
78
79 static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
80                              N_Vector res);
81 static void PrintHeader(realtype rtol, realtype atol);
82 static void PrintOutput(void *mem, realtype t, N_Vector uu);
83 static int check_flag(void *flagvalue, char *funcname, int opt);
84
85 /*
86  *--------------------------------------------------------------------
87  * MAIN PROGRAM
88  *--------------------------------------------------------------------
89  */
90
91 int main()
92 {
93   void *mem;
94   UserData data;
95   N_Vector uu, up, constraints, res;
96   int ier, iout;
97   realtype rtol, atol, t0, t1, tout, tret;
98   long int netf, ncfn, ncfl;
99
100   mem = NULL;
101   data = NULL;
102   uu = up = constraints = res = NULL;
103
104   /* Allocate N-vectors and the user data structure. */
105
106   uu = N_VNew_Serial(NEQ);
```

```
107   if(check_flag((void *)uu, "N_VNew_Serial", 0)) return(1);
108
109   up = N_VNew_Serial(NEQ);
110   if(check_flag((void *)up, "N_VNew_Serial", 0)) return(1);
111
112   res = N_VNew_Serial(NEQ);
113   if(check_flag((void *)res, "N_VNew_Serial", 0)) return(1);
114
115   constraints = N_VNew_Serial(NEQ);
116   if(check_flag((void *)constraints, "N_VNew_Serial", 0)) return(1);
117
118   data = (UserData) malloc(sizeof *data);
119   data->pp = NULL;
120   if(check_flag((void *)data, "malloc", 2)) return(1);
121
122   /* Assign parameters in the user data structure. */
123
124   data->mm  = MGRID;
125   data->dx = ONE/(MGRID-ONE);
126   data->coeff = ONE/(data->dx * data->dx);
127   data->pp = N_VNew_Serial(NEQ);
128   if(check_flag((void *)data->pp, "N_VNew_Serial", 0)) return(1);
129
130   /* Initialize uu, up. */
131
132   SetInitialProfile(data, uu, up, res);
133
134   /* Set constraints to all 1's for nonnegative solution values. */
135
136   N_VConst(ONE, constraints);
137
138   /* Assign various parameters. */
139
140   t0   = ZERO;
141   t1   = RCONST(0.01);
142   rtol = ZERO;
143   atol = RCONST(1.0e-3);
144
145   /* Call IDACreate and IDAMalloc to initialize solution */
146
147   mem = IDACreate();
148   if(check_flag((void *)mem, "IDACreate", 0)) return(1);
149
150   ier = IDASetRdata(mem, data);
151   if(check_flag(&ier, "IDASetRdata", 1)) return(1);
152
153   ier = IDASetConstraints(mem, constraints);
154   if(check_flag(&ier, "IDASetConstraints", 1)) return(1);
155
156   ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, &rtol, &atol);
157   if(check_flag(&ier, "IDAMalloc", 1)) return(1);
158
159   /* Call IDASpgmr to specify the linear solver. */
160
```

```
  ier = IDASpgmr(mem, 0);
  if(check_flag(&ier, "IDASpgmr", 1)) return(1);

  ier = IDASpgmrSetPrecSetupFn(mem, PsetupHeat);
  if(check_flag(&ier, "IDASpgmrSetPrecSetupFn", 1)) return(1);

  ier = IDASpgmrSetPrecSolveFn(mem, PsolveHeat);
  if(check_flag(&ier, "IDASpgmrSetPrecSolveFn", 1)) return(1);

  ier = IDASpgmrSetPrecData(mem, data);
  if(check_flag(&ier, "IDASpgmrSetPrecData", 1)) return(1);

  /* Print output heading. */
  PrintHeader(rtol, atol);

  /*
   * ------------------------------------------------------------------------
   * CASE I
   * ------------------------------------------------------------------------
   */

  /* Print case number, output table heading, and initial line of table. */

  printf("\n\nCase 1: gsytpe = MODIFIED_GS\n");
  printf("\n   Output Summary (umax = max-norm of solution) \n\n");
  printf("  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps\n" );
  printf("-----------------------------------------------------------------------\n");
  PrintOutput(mem,t0,uu);

  /* Loop over output times, call IDASolve, and print results. */

  for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
    ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
    if(check_flag(&ier, "IDASolve", 1)) return(1);
    PrintOutput(mem, tret, uu);
  }

  /* Print remaining counters. */

  ier = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&ier, "IDAGetNumErrTestFails", 1);

  ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);

  ier = IDASpgmrGetNumConvFails(mem, &ncfl);
  check_flag(&ier, "IDASpgmrGetNumConvFails", 1);

  printf("\nError test failures            = %ld\n", netf);
  printf("Nonlinear convergence failures = %ld\n", ncfn);
  printf("Linear convergence failures    = %ld\n", ncfl);

  /*
   * ------------------------------------------------------------------------
```

```
215      * CASE II
216      * ------------------------------------------------------------------------
217      */
218
219     /* Re-initialize uu, up. */
220
221     SetInitialProfile(data, uu, up, res);
222
223     /* Re-initialize IDA and IDASPGMR */
224
225     ier = IDAReInit(mem, resHeat, t0, uu, up, IDA_SS, &rtol, &atol);
226     if(check_flag(&ier, "IDAReInit", 1)) return(1);
227
228     ier = IDASpgmrSetGSType(mem, CLASSICAL_GS);
229     if(check_flag(&ier, "IDASpgmrSetGSType",1)) return(1);
230
231     /* Print case number, output table heading, and initial line of table. */
232
233     printf("\n\nCase 2: gstype = CLASSICAL_GS\n");
234     printf("\n   Output Summary (umax = max-norm of solution) \n\n");
235     printf("  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps\n" );
236     printf("------------------------------------------------------------------------\n");
237     PrintOutput(mem,t0,uu);
238
239     /* Loop over output times, call IDASolve, and print results. */
240
241     for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
242       ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
243       if(check_flag(&ier, "IDASolve", 1)) return(1);
244       PrintOutput(mem, tret, uu);
245     }
246
247     /* Print remaining counters. */
248
249     ier = IDAGetNumErrTestFails(mem, &netf);
250     check_flag(&ier, "IDAGetNumErrTestFails", 1);
251
252     ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
253     check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
254
255     ier = IDASpgmrGetNumConvFails(mem, &ncfl);
256     check_flag(&ier, "IDASpgmrGetNumConvFails", 1);
257
258     printf("\nError test failures            = %ld\n", netf);
259     printf("Nonlinear convergence failures = %ld\n", ncfn);
260     printf("Linear convergence failures    = %ld\n", ncfl);
261
262     /* Free Memory */
263
264     IDAFree(mem);
265
266     N_VDestroy_Serial(uu);
267     N_VDestroy_Serial(up);
268     N_VDestroy_Serial(constraints);
```

```
269   N_VDestroy_Serial(res);
270
271   N_VDestroy_Serial(data->pp);
272   free(data);
273
274   return(0);
275 }
276
277 /*
278  *--------------------------------------------------------------------
279  * FUNCTIONS CALLED BY IDA
280  *--------------------------------------------------------------------
281  */
282
283 /*
284  * resHeat: heat equation system residual function (user-supplied)
285  * This uses 5-point central differencing on the interior points, and
286  * includes algebraic equations for the boundary values.
287  * So for each interior point, the residual component has the form
288  *    res_i = u'_i - (central difference)_i
289  * while for each boundary point, it is res_i = u_i.
290  */
291
292 int resHeat(realtype tt,
293             N_Vector uu, N_Vector up, N_Vector rr,
294             void *res_data)
295 {
296   long int i, j, offset, loc, mm;
297   realtype *uu_data, *up_data, *rr_data, coeff, dif1, dif2;
298   UserData data;
299
300   uu_data = NV_DATA_S(uu);
301   up_data = NV_DATA_S(up);
302   rr_data = NV_DATA_S(rr);
303
304   data = (UserData) res_data;
305
306   coeff = data->coeff;
307   mm    = data->mm;
308
309   /* Initialize rr to uu, to take care of boundary equations. */
310   N_VScale(ONE, uu, rr);
311
312   /* Loop over interior points; set res = up - (central difference). */
313   for (j = 1; j < MGRID-1; j++) {
314     offset = mm*j;
315     for (i = 1; i < mm-1; i++) {
316       loc = offset + i;
317       dif1 = uu_data[loc-1]  + uu_data[loc+1]  - TWO * uu_data[loc];
318       dif2 = uu_data[loc-mm] + uu_data[loc+mm] - TWO * uu_data[loc];
319       rr_data[loc]= up_data[loc] - coeff * ( dif1 + dif2 );
320     }
321   }
322
```

```
  return(0);
}

/*
 * PsetupHeat: setup for diagonal preconditioner for iheatsk.
 *
 * The optional user-supplied functions PsetupHeat and
 * PsolveHeat together must define the left preconditoner
 * matrix P approximating the system Jacobian matrix
 *                   J = dF/du + cj*dF/du'
 * (where the DAE system is F(t,u,u') = 0), and solve the linear
 * systems P z = r.   This 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 and data (user data structure, with
 * pp etc.) are used from the PsetupdHeat argument list.
 */

int PsetupHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{

  long int i, j, offset, loc, mm;
  realtype *ppv, pelinv;
  UserData data;

  data = (UserData) prec_data;
  ppv = NV_DATA_S(data->pp);
  mm = data->mm;

  /* Initialize the entire vector to 1., then set the interior points to the
     correct value for preconditioning. */
  N_VConst(ONE,data->pp);

  /* Compute the inverse of the preconditioner diagonal elements. */
  pelinv = ONE/(c_j + FOUR*data->coeff);

  for (j = 1; j < mm-1; j++) {
    offset = mm * j;
    for (i = 1; i < mm-1; i++) {
      loc = offset + i;
      ppv[loc] = pelinv;
    }
  }

  return(0);
}

/*
 * PsolveHeat: solve preconditioner linear system.
```

```
 * This routine multiplies the input vector rvec by the vector pp
 * containing the inverse diagonal Jacobian elements (previously
 * computed in PrecondHeateq), returning the result in zvec.
 */

int PsolveHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               N_Vector rvec, N_Vector zvec,
               realtype c_j, realtype delta, void *prec_data,
               N_Vector tmp)
{
  UserData data;
  data = (UserData) prec_data;
  N_VProd(data->pp, rvec, zvec);
  return(0);
}

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

/*
 * SetInitialProfile: routine to initialize u and up vectors.
 */

static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
                             N_Vector res)
{
  long int mm, mm1, i, j, offset, loc;
  realtype xfact, yfact, *udata, *updata;

  mm = data->mm;

  udata = NV_DATA_S(uu);
  updata = NV_DATA_S(up);

  /* Initialize uu on all grid points. */
  mm1 = mm - 1;
  for (j = 0; j < mm; j++) {
    yfact = data->dx * j;
    offset = mm*j;
    for (i = 0;i < mm; i++) {
      xfact = data->dx * i;
      loc = offset + i;
      udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
    }
  }

  /* Initialize up vector to 0. */
  N_VConst(ZERO, up);

  /* resHeat sets res to negative of ODE RHS values at interior points. */
```

```
  resHeat(ZERO, uu, up, res, data);

  /* Copy -res into up to get correct interior initial up values. */
  N_VScale(-ONE, res, up);

  /* Set up at boundary points to zero. */
  for (j = 0; j < mm; j++) {
    offset = mm*j;
    for (i = 0; i < mm; i++) {
      loc = offset + i;
      if (j == 0 || j == mm1 || i == 0 || i == mm1 ) updata[loc] = ZERO;
    }
  }

  return(0);
  }

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

static void PrintHeader(realtype rtol, realtype atol)
{
  printf("\niheatsk: Heat equation, serial example problem for IDA \n");
  printf("         Discretized heat equation on 2D unit square. \n");
  printf("         Zero boundary conditions,");
  printf(" polynomial initial conditions.\n");
  printf("         Mesh dimensions: %d x %d", MGRID, MGRID);
  printf("        Total system size: %d\n\n", NEQ);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Constraints set to force all solution components >= 0. \n");
  printf("Linear solver: IDASPGMR, preconditioner using diagonal elements. \n");
}

/*
 * PrintOutput: print max norm of solution and current solver statistics
 */

static void PrintOutput(void *mem, realtype t, N_Vector uu)
{
  realtype hused, umax;
  long int nst, nni, nje, nre, nreS, nli, npe, nps;
  int kused, ier;

  umax = N_VMaxNorm(uu);

  ier = IDAGetLastOrder(mem, &kused);
  check_flag(&ier, "IDAGetLastOrder", 1);
```

```
485   ier = IDAGetNumSteps(mem, &nst);
486   check_flag(&ier, "IDAGetNumSteps", 1);
487   ier = IDAGetNumNonlinSolvIters(mem, &nni);
488   check_flag(&ier, "IDAGetNumNonlinSolvIters", 1);
489   ier = IDAGetNumResEvals(mem, &nre);
490   check_flag(&ier, "IDAGetNumResEvals", 1);
491   ier = IDAGetLastStep(mem, &hused);
492   check_flag(&ier, "IDAGetLastStep", 1);
493   ier = IDASpgmrGetNumJtimesEvals(mem, &nje);
494   check_flag(&ier, "IDASpgmrGetNumJtimesEvals", 1);
495   ier = IDASpgmrGetNumLinIters(mem, &nli);
496   check_flag(&ier, "IDASpgmrGetNumLinIters", 1);
497   ier = IDASpgmrGetNumResEvals(mem, &nreS);
498   check_flag(&ier, "IDASpgmrGetNumResEvals", 1);
499   ier = IDASpgmrGetNumPrecEvals(mem, &npe);
500   check_flag(&ier, "IDASpgmrGetPrecEvals", 1);
501   ier = IDASpgmrGetNumPrecSolves(mem, &nps);
502   check_flag(&ier, "IDASpgmrGetNumPrecSolves", 1);
503 
504 #if defined(SUNDIALS_EXTENDED_PRECISION)
505   printf(" %5.2Lf %13.5Le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2Le  %3ld %3ld\n",
506          t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
507 #elif defined(SUNDIALS_DOUBLE_PRECISION)
508   printf(" %5.2f %13.5le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2le  %3ld %3ld\n",
509          t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
510 #else
511   printf(" %5.2f %13.5e  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2e  %3ld %3ld\n",
512          t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
513 #endif
514 }
515 
516 /*
517  * Check function return value...
518  *   opt == 0 means SUNDIALS function allocates memory so check if
519  *            returned NULL pointer
520  *   opt == 1 means SUNDIALS function returns a flag so check if
521  *            flag >= 0
522  *   opt == 2 means function allocates memory so check if returned
523  *            NULL pointer
524  */
525 
526 static int check_flag(void *flagvalue, char *funcname, int opt)
527 {
528   int *errflag;
529 
530   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
531   if (opt == 0 && flagvalue == NULL) {
532     fprintf(stderr,
533             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
534             funcname);
535     return(1);
536   } else if (opt == 1) {
537     /* Check if flag < 0 */
538     errflag = (int *) flagvalue;
```

```
539       if (*errflag < 0) {
540         fprintf(stderr,
541                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
542                 funcname, *errflag);
543         return(1);
544       }
545     } else if (opt == 2 && flagvalue == NULL) {
546       /* Check if function returned NULL pointer - no memory allocated */
547       fprintf(stderr,
548               "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
549               funcname);
550       return(1);
551     }
552
553     return(0);
554   }
```

D Listing of `iheatpk.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.15.2.1 $
 * $Date: 2005/03/17 22:50:50 $
 * -----------------------------------------------------------------
 * 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 "sundialstypes.h"
#include "sundialsmath.h"
#include "nvector_parallel.h"
#include "ida.h"
#include "idaspgmr.h"
#include "iterative.h"
#include "mpi.h"

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

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

```
53
54 #define NPEX         2              /* No. PEs in x direction of PE array */
55 #define NPEY         2              /* No. PEs in y direction of PE array */
56                                     /* Total no. PEs = NPEX*NPEY */
57 #define MXSUB        5              /* No. x points per subgrid */
58 #define MYSUB        5              /* No. y points per subgrid */
59
60 #define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
61 #define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
62                                     /* Spatial mesh is MX by MY */
63
64 typedef struct {
65   long int thispe, mx, my, ixsub, jysub, npex, npey, mxsub, mysub;
66   realtype    dx, dy, coeffx, coeffy, coeffxy;
67   realtype    uext[(MXSUB+2)*(MYSUB+2)];
68   N_Vector    pp;    /* vector of diagonal preconditioner elements */
69   MPI_Comm    comm;
70 } *UserData;
71
72 /* User-supplied residual function and supporting routines */
73
74 int resHeat(realtype tt,
75             N_Vector uu, N_Vector up, N_Vector rr,
76             void *res_data);
77
78 static int rescomm(N_Vector uu, N_Vector up, void *res_data);
79
80 static int reslocal(realtype tt, N_Vector uu, N_Vector up,
81                     N_Vector res,  void *res_data);
82
83 static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
84                  long int dsizex, long int dsizey, realtype uarray[]);
85
86 static int BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
87                      long int ixsub, long int jysub,
88                      long int dsizex, long int dsizey,
89                      realtype uext[], realtype buffer[]);
90
91 static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
92                      long int dsizex, realtype uext[], realtype buffer[]);
93
94 /* User-supplied preconditioner routines */
95
96 int PsolveHeat(realtype tt,
97                N_Vector uu, N_Vector up, N_Vector rr,
98                N_Vector rvec, N_Vector zvec,
99                realtype c_j, realtype delta, void *prec_data,
100               N_Vector tmp);
101
102 int PsetupHeat(realtype tt,
103               N_Vector yy, N_Vector yp, N_Vector rr,
104               realtype c_j, void *prec_data,
105               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);
106
```

```
/* Private function to check function return values */

static int InitUserData(int thispe, MPI_Comm comm, UserData data);

static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
                             N_Vector res, UserData data);

static void PrintHeader(long int Neq, realtype rtol, realtype atol);

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();
```

```
215     if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);
216
217     ier = IDASetRdata(mem, data);
218     if(check_flag(&ier, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);
219
220     ier = IDASetSuppressAlg(mem, TRUE);
221     if(check_flag(&ier, "IDASetSuppressAlg", 1, thispe)) MPI_Abort(comm, 1);
222
223     ier = IDASetId(mem, id);
224     if(check_flag(&ier, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);
225
226     ier = IDASetConstraints(mem, constraints);
227     if(check_flag(&ier, "IDASetConstraints", 1, thispe)) MPI_Abort(comm, 1);
228
229     ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, &rtol, &atol);
230     if(check_flag(&ier, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
231
232     /* Call IDASpgmr to specify the linear solver. */
233
234     ier = IDASpgmr(mem, 0);
235     if(check_flag(&ier, "IDASpgmr", 1, thispe)) MPI_Abort(comm, 1);
236
237     ier = IDASpgmrSetPrecSetupFn(mem, PsetupHeat);
238     if(check_flag(&ier, "IDASpgmrSetPrecSetupFn", 1, thispe)) MPI_Abort(comm, 1);
239
240     ier = IDASpgmrSetPrecSolveFn(mem, PsolveHeat);
241     if(check_flag(&ier, "IDASpgmrSetPrecSolveFn", 1, thispe)) MPI_Abort(comm, 1);
242
243     ier = IDASpgmrSetPrecData(mem, data);
244     if(check_flag(&ier, "IDASpgmrSetPrecData", 1, thispe)) MPI_Abort(comm, 1);
245
246     /* Print output heading (on processor 0 only) and intial solution  */
247
248     if (thispe == 0) PrintHeader(Neq, rtol, atol);
249     PrintOutput(thispe, mem, t0, uu);
250
251     /* Loop over tout, call IDASolve, print output. */
252
253     for (tout = t1, iout = 1; iout <= NOUT; iout++, tout *= TWO) {
254
255       ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
256       if(check_flag(&ier, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
257
258       PrintOutput(thispe, mem, tret, uu);
259
260     }
261
262     /* Print remaining counters. */
263
264     if (thispe == 0) PrintFinalStats(mem);
265
266     /* Free memory */
267
268     IDAFree(mem);
```

```
269
270   N_VDestroy_Parallel(id);
271   N_VDestroy_Parallel(constraints);
272   N_VDestroy_Parallel(res);
273   N_VDestroy_Parallel(up);
274   N_VDestroy_Parallel(uu);
275
276   N_VDestroy_Parallel(data->pp);
277   free(data);
278
279   MPI_Finalize();
280
281   return(0);
282
283 }
284
285 /*
286  *--------------------------------------------------------------------
287  * FUNCTIONS CALLED BY IDA
288  *--------------------------------------------------------------------
289  */
290
291 /*
292  * resHeat: heat equation system residual function
293  * This uses 5-point central differencing on the interior points, and
294  * includes algebraic equations for the boundary values.
295  * So for each interior point, the residual component has the form
296  *    res_i = u'_i - (central difference)_i
297  * while for each boundary point, it is res_i = u_i.
298  *
299  * This parallel implementation uses several supporting routines.
300  * First a call is made to rescomm to do communication of subgrid boundary
301  * data into array uext.  Then reslocal is called to compute the residual
302  * on individual processors and their corresponding domains.  The routines
303  * BSend, BRecvPost, and BREcvWait handle interprocessor communication
304  * of uu required to calculate the residual.
305  */
306
307 int resHeat(realtype tt,
308             N_Vector uu, N_Vector up, N_Vector rr,
309             void *res_data)
310 {
311   int retval;
312
313   /* Call rescomm to do inter-processor communication. */
314   retval = rescomm(uu, up, res_data);
315
316   /* Call reslocal to calculate res. */
317   retval = reslocal(tt, uu, up, rr, res_data);
318
319   return(0);
320
321 }
322
```

```
/*
 * PsetupHeat: setup for diagonal preconditioner for heatsk.
 *
 * The optional user-supplied functions PsetupHeat and
 * PsolveHeat together must define the left preconditoner
 * matrix P approximating the system Jacobian matrix
 *                   J = dF/du + cj*dF/du'
 * (where the DAE system is F(t,u,u') = 0), and solve the linear
 * systems P z = r.   This 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 and data (user data structure, with
 * pp etc.) are used from the PsetupHeat argument list.
 *
 */

int PsetupHeat(realtype tt,
               N_Vector yy, N_Vector yp, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  realtype *ppv, pelinv;
  long int lx, ly, ixbegin, ixend, jybegin, jyend, locu, mxsub, mysub;
  long int ixsub, jysub, npex, npey;
  UserData data;

  data = (UserData) prec_data;

  ppv = NV_DATA_P(data->pp);
  ixsub = data->ixsub;
  jysub = data->jysub;
  mxsub = data->mxsub;
  mysub = data->mysub;
  npex  = data->npex;
  npey  = data->npey;

  /* Initially set all pp elements to one. */
  N_VConst(ONE, data->pp);

  /* Prepare to loop over subgrid. */
  ixbegin = 0;
  ixend   = mxsub-1;
  jybegin = 0;
  jyend   = mysub-1;
  if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
  if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
  pelinv = ONE/(c_j + data->coeffxy);

  /* Load the inverse of the preconditioner diagonal elements
     in loop over all the local subgrid. */

  for (ly = jybegin; ly <=jyend; ly++) {
```

```
377       for (lx = ixbegin; lx <= ixend; lx++) {
378         locu  = lx + ly*mxsub;
379         ppv[locu] = pelinv;
380       }
381     }
382 
383     return(0);
384 
385   }
386 
387   /*
388    * PsolveHeat: solve preconditioner linear system.
389    * This routine multiplies the input vector rvec by the vector pp
390    * containing the inverse diagonal Jacobian elements (previously
391    * computed in PsetupHeat), returning the result in zvec.
392    */
393 
394   int PsolveHeat(realtype tt,
395                  N_Vector uu, N_Vector up, N_Vector rr,
396                  N_Vector rvec, N_Vector zvec,
397                  realtype c_j, realtype delta, void *prec_data,
398                  N_Vector tmp)
399   {
400     UserData data;
401 
402     data = (UserData) prec_data;
403 
404     N_VProd(data->pp, rvec, zvec);
405 
406     return(0);
407 
408   }
409 
410   /*
411    *--------------------------------------------------------------------
412    * SUPPORTING FUNCTIONS
413    *--------------------------------------------------------------------
414    */
415 
416 
417   /*
418    * rescomm routine.  This routine performs all inter-processor
419    * communication of data in u needed to calculate G.
420    */
421 
422   static int rescomm(N_Vector uu, N_Vector up, void *res_data)
423   {
424     UserData data;
425     realtype *uarray, *uext, buffer[2*MYSUB];
426     MPI_Comm comm;
427     long int thispe, ixsub, jysub, mxsub, mysub;
428     MPI_Request request[4];
429 
430     data = (UserData) res_data;
```

```
  uarray = NV_DATA_P(uu);

  /* Get comm, thispe, subgrid indices, data sizes, extended array uext. */
  comm = data->comm;  thispe = data->thispe;
  ixsub = data->ixsub;   jysub = data->jysub;
  mxsub = data->mxsub;   mysub = data->mysub;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs. */
  BRecvPost(comm, request, thispe, ixsub, jysub, mxsub, mysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs. */
  BSend(comm, thispe, ixsub, jysub, mxsub, mysub, uarray);

  /* Finish receiving boundary data from neighboring PEs. */
  BRecvWait(request, ixsub, jysub, mxsub, uext, buffer);

  return(0);

}

/*
 * reslocal routine.  Compute res = F(t, uu, up).  This routine assumes
 * that all inter-processor communication of data needed to calculate F
 * has already been done, and that this data is in the work array uext.
 */

static int reslocal(realtype tt,
                    N_Vector uu, N_Vector up, N_Vector rr,
                    void *res_data)
{
  realtype *uext, *uuv, *upv, *resv;
  realtype termx, termy, termctr;
  long int lx, ly, offsetu, offsetue, locu, locue;
  long int ixsub, jysub, mxsub, mxsub2, mysub, npex, npey;
  long int ixbegin, ixend, jybegin, jyend;
  UserData data;

  /* Get subgrid indices, array sizes, extended work array uext. */

  data = (UserData) res_data;
  uext = data->uext;
  uuv = NV_DATA_P(uu);
  upv = NV_DATA_P(up);
  resv = NV_DATA_P(rr);
  ixsub = data->ixsub; jysub = data->jysub;
  mxsub = data->mxsub; mxsub2 = data->mxsub + 2;
  mysub = data->mysub; npex = data->npex; npey = data->npey;

  /* Initialize all elements of rr to uu. This sets the boundary
     elements simply without indexing hassles. */

  N_VScale(ONE, uu, rr);

```

```
485   /* Copy local segment of u vector into the working extended array uext.
486      This completes uext prior to the computation of the rr vector.     */
487
488   offsetu = 0;
489   offsetue = mxsub2 + 1;
490   for (ly = 0; ly < mysub; ly++) {
491     for (lx = 0; lx < mxsub; lx++) uext[offsetue+lx] = uuv[offsetu+lx];
492     offsetu = offsetu + mxsub;
493     offsetue = offsetue + mxsub2;
494   }
495
496   /* Set loop limits for the interior of the local subgrid. */
497
498   ixbegin = 0;
499   ixend   = mxsub-1;
500   jybegin = 0;
501   jyend   = mysub-1;
502   if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
503   if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
504
505   /* Loop over all grid points in local subgrid. */
506
507   for (ly = jybegin; ly <=jyend; ly++) {
508     for (lx = ixbegin; lx <= ixend; lx++) {
509       locu  = lx + ly*mxsub;
510       locue = (lx+1) + (ly+1)*mxsub2;
511       termx = data->coeffx *(uext[locue-1]      + uext[locue+1]);
512       termy = data->coeffy *(uext[locue-mxsub2] + uext[locue+mxsub2]);
513       termctr = data->coeffxy*uext[locue];
514       resv[locu] = upv[locu] - (termx + termy - termctr);
515    }
516   }
517   return(0);
518
519 }
520
521 /*
522  * Routine to send boundary data to neighboring PEs.
523  */
524
525 static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
526                  long int dsizex, long int dsizey, realtype uarray[])
527 {
528   long int ly, offsetu;
529   realtype bufleft[MYSUB], bufright[MYSUB];
530
531   /* If jysub > 0, send data from bottom x-line of u. */
532
533   if (jysub != 0)
534     MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, thispe-NPEX, 0, comm);
535
536   /* If jysub < NPEY-1, send data from top x-line of u. */
537
538   if (jysub != NPEY-1) {
```

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

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

```
647       for (ly = 0; ly < MYSUB; ly++) {
648         offsetue = (ly+1)*dsizex2;
649         uext[offsetue] = bufleft[ly];
650       }
651     }
652 
653     /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
654     if (ixsub != NPEX-1) {
655       MPI_Wait(&request[3],&status);
656 
657       /* Copy the buffer to uext */
658       for (ly = 0; ly < MYSUB; ly++) {
659         offsetue = (ly+2)*dsizex2 - 1;
660         uext[offsetue] = bufright[ly];
661       }
662     }
663 
664     return(0);
665 
666   }
667 
668   /*
669    *--------------------------------------------------------------------
670    * PRIVATE FUNCTIONS
671    *--------------------------------------------------------------------
672    */
673 
674   /*
675    * InitUserData initializes the user's data block data.
676    */
677 
678   static int InitUserData(int thispe, MPI_Comm comm, UserData data)
679   {
680     data->thispe = thispe;
681     data->dx = ONE/(MX-ONE);       /* Assumes a [0,1] interval in x. */
682     data->dy = ONE/(MY-ONE);       /* Assumes a [0,1] interval in y. */
683     data->coeffx  = ONE/(data->dx * data->dx);
684     data->coeffy  = ONE/(data->dy * data->dy);
685     data->coeffxy = TWO/(data->dx * data->dx) + TWO/(data->dy * data->dy) ;
686     data->jysub   = thispe/NPEX;
687     data->ixsub   = thispe - data->jysub * NPEX;
688     data->npex    = NPEX;
689     data->npey    = NPEY;
690     data->mx      = MX;
691     data->my      = MY;
692     data->mxsub = MXSUB;
693     data->mysub = MYSUB;
694     data->comm    = comm;
695     return(0);
696 
697   }
698 
699   /*
700    * 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. */

  udata = NV_DATA_P(uu);
  iddata = NV_DATA_P(id);

  /* Set mesh spacings and subgrid indices for this PE. */
  dx = data->dx;
  dy = data->dy;
  ixsub = data->ixsub;
  jysub = data->jysub;

  /* Set beginning and ending locations in the global array corresponding
     to the portion of that array assigned to this processor. */
  ixbegin = MXSUB*ixsub;
  ixend   = MXSUB*(ixsub+1) - 1;
  jybegin = MYSUB*jysub;
  jyend   = MYSUB*(jysub+1) - 1;

  /* Loop over the local array, computing the initial profile value.
     The global indices are (i,j) and the local indices are (iloc,jloc).
     Also set the id vector to zero for boundary points, one otherwise. */

  N_VConst(ONE,id);
  for (j = jybegin, jloc = 0; j <= jyend; j++, jloc++) {
    yfact = data->dy*j;
    offset= jloc*MXSUB;
    for (i = ixbegin, iloc = 0; i <= ixend; i++, iloc++) {
      xfact = data->dx * i;
      loc = offset + iloc;
      udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
      if (i == 0 || i == MX-1 || j == 0 || j == MY-1) iddata[loc] = ZERO;
    }
  }

  /* Initialize up. */

  N_VConst(ZERO, up);    /* Initially set up = 0. */

  /* resHeat sets res to negative of ODE RHS values at interior points. */
  resHeat(ZERO, uu, up, res, data);

  /* Copy -res into up to get correct initial up values. */
  N_VScale(-ONE, res, up);

  return(0);
```

```
}

/*
 * Print first lines of output and table heading
 */

static void PrintHeader(long int Neq, realtype rtol, realtype atol)
{
  printf("\niheatpk: Heat equation, parallel example problem for IDA \n");
  printf("         Discretized heat equation on 2D unit square. \n");
  printf("         Zero boundary conditions,");
  printf(" polynomial initial conditions.\n");
  printf("         Mesh dimensions: %d x %d", MX, MY);
  printf("        Total system size: %ld\n\n", Neq);
  printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
  printf("        Processor array: %d x %d\n", NPEX, NPEY);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Constraints set to force all solution components >= 0. \n");
  printf("SUPPRESSALG = TRUE to suppress local error testing on ");
  printf("all boundary components. \n");
  printf("Linear solver: IDASPGMR  ");
  printf("Preconditioner: diagonal elements only.\n");

  /* Print output table heading and initial line of table. */
  printf("\n   Output Summary (umax = max-norm of solution) \n\n");
  printf("  time     umax       k  nst  nni  nli   nre   nreS     h      npe nps\n");
  printf("-----------------------------------------------------------------------\n");
}

/*
 * PrintOutput: print max norm of solution and current solver statistics
 */

static void PrintOutput(int id, void *mem, realtype t, N_Vector uu)
{
  realtype hused, umax;
  long int nst, nni, nje, nre, nreS, nli, npe, nps;
  int kused, ier;

  umax = N_VMaxNorm(uu);

  if (id == 0) {

    ier = IDAGetLastOrder(mem, &kused);
    check_flag(&ier, "IDAGetLastOrder", 1, id);
    ier = IDAGetNumSteps(mem, &nst);
    check_flag(&ier, "IDAGetNumSteps", 1, id);
    ier = IDAGetNumNonlinSolvIters(mem, &nni);
```

```
809       check_flag(&ier, "IDAGetNumNonlinSolvIters", 1, id);
810       ier = IDAGetNumResEvals(mem, &nre);
811       check_flag(&ier, "IDAGetNumResEvals", 1, id);
812       ier = IDAGetLastStep(mem, &hused);
813       check_flag(&ier, "IDAGetLastStep", 1, id);
814       ier = IDASpgmrGetNumJtimesEvals(mem, &nje);
815       check_flag(&ier, "IDASpgmrGetNumJtimesEvals", 1, id);
816       ier = IDASpgmrGetNumLinIters(mem, &nli);
817       check_flag(&ier, "IDASpgmrGetNumLinIters", 1, id);
818       ier = IDASpgmrGetNumResEvals(mem, &nreS);
819       check_flag(&ier, "IDASpgmrGetNumResEvals", 1, id);
820       ier = IDASpgmrGetNumPrecEvals(mem, &npe);
821       check_flag(&ier, "IDASpgmrGetPrecEvals", 1, id);
822       ier = IDASpgmrGetNumPrecSolves(mem, &nps);
823       check_flag(&ier, "IDASpgmrGetNumPrecSolves", 1, id);
824
825   #if defined(SUNDIALS_EXTENDED_PRECISION)
826       printf(" %5.2Lf %13.5Le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2Le  %3ld %3ld\n",
827              t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
828   #elif defined(SUNDIALS_DOUBLE_PRECISION)
829       printf(" %5.2f %13.5le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2le  %3ld %3ld\n",
830              t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
831   #else
832       printf(" %5.2f %13.5e  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2e  %3ld %3ld\n",
833              t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
834   #endif
835
836     }
837   }
838
839   /*
840    * Print some final integrator statistics
841    */
842
843   static void PrintFinalStats(void *mem)
844   {
845     long int netf, ncfn, ncfl;
846
847     IDAGetNumErrTestFails(mem, &netf);
848     IDAGetNumNonlinSolvConvFails(mem, &ncfn);
849     IDASpgmrGetNumConvFails(mem, &ncfl);
850
851     printf("\nError test failures            = %ld\n", netf);
852     printf("Nonlinear convergence failures = %ld\n", ncfn);
853     printf("Linear convergence failures    = %ld\n", ncfl);
854   }
855
856   /*
857    * Check function return value...
858    *   opt == 0 means SUNDIALS function allocates memory so check if
859    *            returned NULL pointer
860    *   opt == 1 means SUNDIALS function returns a flag so check if
861    *            flag >= 0
862    *   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);
}
```

E Listing of `iwebbbd.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.21.2.1 $
 * $Date: 2005/03/17 22:50:50 $
 * -----------------------------------------------------------------
 * 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 "sundialstypes.h"    /* Definitions of realtype and booleantype       */
#include "iterative.h"        /* Contains the types of preconditioning         */
#include "ida.h"              /* Main header file                              */
#include "idaspgmr.h"         /* Use IDASPGMR linear solver                    */
#include "nvector_parallel.h" /* Definitions of type N_Vector, macro NV_DATA_P */
```

```
#include "sundialsmath.h"     /* Contains RSqrt routine                          */
#include "smalldense.h"       /* Contains definitions for denalloc routine       */
#include "mpi.h"              /* MPI library routines                            */
#include "idabbdpre.h"        /* Definitions for the IDABBDPRE preconditioner    */

/* Problem Constants. */

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

#define PI          RCONST(3.1415926535898) /* pi */
#define FOURPI      (RCONST(4.0)*PI)        /* 4 pi */

#define MXSUB       10    /* Number of x mesh points per processor subgrid */
#define MYSUB       10    /* Number of y mesh points per processor subgrid */
#define NPEX        2     /* Number of subgrids in the x direction */
#define NPEY        2     /* Number of subgrids in the y direction */
#define MX          (MXSUB*NPEX)      /* MX = number of x mesh points */
#define MY          (MYSUB*NPEY)      /* MY = number of y mesh points */
#define NSMXSUB     (NUM_SPECIES * MXSUB)
#define NEQ         (NUM_SPECIES*MX*MY) /* Number of equations in system */
#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)  /*  rtol tolerance */
#define ATOL        RCONST(1.e-5)  /*  atol tolerance */
#define ZERO        RCONST(0.)     /* 0. */
#define ONE         RCONST(1.0)    /* 1. */
#define NOUT        6
#define TMULT       RCONST(10.0)   /* Multiplier for tout values */
#define TADD        RCONST(0.3)    /* Increment for tout values */

/* User-defined vector 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_P(vv, (i)*NUM_SPECIES + (j)*NSMXSUB ))

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

typedef struct {
  long int ns, np, thispe, npes, ixsub, jysub, npex, npey;
```

```
  long int mxsub, mysub, nsmxsub, nsmxsub2;
  realtype dx, dy, **acoef;
  realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES],
    rhs[NUM_SPECIES], cext[(MXSUB+2)*(MYSUB+2)*NUM_SPECIES];
  MPI_Comm comm;
  N_Vector rates;
  long int n_local;
} *UserData;

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

static int resweb(realtype tt,
                  N_Vector cc, N_Vector cp, N_Vector rr,
                  void *res_data);

static int reslocal(long int Nlocal, realtype tt,
                    N_Vector cc, N_Vector cp, N_Vector res,
                    void *res_data);

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

/* Prototypes for supporting functions */

static void BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
                  long int dsizex, long int dsizey, realtype carray[]);

static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
                      long int ixsub, long int jysub,
                      long int dsizex, long int dsizey,
                      realtype cext[], realtype buffer[]);

static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
                      long int dsizex, realtype cext[], realtype buffer[]);

static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata);

static realtype dotprod(long int size, realtype *x1, realtype *x2);

/* Prototypes for private functions */

static void InitUserData(UserData webdata, int thispe, int npes,
                         MPI_Comm comm);

static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               N_Vector scrtch, UserData webdata);

static void PrintHeader(long int SystemSize, int maxl,
                        long int mudq, long int mldq,
                        long int mukeep, long int mlkeep,
                        realtype rtol, realtype atol);

```

```
215 static void PrintOutput(void *mem, N_Vector cc, realtype time,
216                         UserData webdata, MPI_Comm comm);
217
218 static void PrintFinalStats(void *mem, void *P_data);
219
220 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
221
222 /*
223  *--------------------------------------------------------------------
224  * MAIN PROGRAM
225  *--------------------------------------------------------------------
226  */
227
228 int main(int argc, char *argv[])
229 {
230   MPI_Comm comm;
231   void *mem, *P_data;
232   UserData webdata;
233   long int SystemSize, local_N, mudq, mldq, mukeep, mlkeep;
234   realtype rtol, atol, t0, tout, tret;
235   N_Vector cc, cp, res, id;
236   int thispe, npes, maxl, iout, retval;
237
238   cc = cp = res = id = NULL;
239   webdata = NULL;
240   mem = P_data = NULL;
241
242   /* Set communicator, and get processor number and total number of PE's. */
243
244   MPI_Init(&argc, &argv);
245   comm = MPI_COMM_WORLD;
246   MPI_Comm_rank(comm, &thispe);
247   MPI_Comm_size(comm, &npes);
248
249   if (npes != NPEX*NPEY) {
250     if (thispe == 0)
251       fprintf(stderr,
252               "\nMPI_ERROR(0): npes = %d not equal to NPEX*NPEY = %d\n",
253               npes, NPEX*NPEY);
254     MPI_Finalize();
255     return(1);
256   }
257
258   /* Set local length (local_N) and global length (SystemSize). */
259
260   local_N = MXSUB*MYSUB*NUM_SPECIES;
261   SystemSize = NEQ;
262
263   /* Set up user data block webdata. */
264
265   webdata = (UserData) malloc(sizeof *webdata);
266   webdata->rates = N_VNew_Parallel(comm, local_N, SystemSize);
267   webdata->acoef = denalloc(NUM_SPECIES);
268
```

```
269   InitUserData(webdata, thispe, npes, comm);
270
271   /* Create needed vectors, and load initial values.
272      The vector res is used temporarily only.        */
273
274   cc  = N_VNew_Parallel(comm, local_N, SystemSize);
275   if(check_flag((void *)cc, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
276
277   cp  = N_VNew_Parallel(comm, local_N, SystemSize);
278   if(check_flag((void *)cp, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
279
280   res = N_VNew_Parallel(comm, local_N, SystemSize);
281   if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
282
283   id  = N_VNew_Parallel(comm, local_N, SystemSize);
284   if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);
285
286   SetInitialProfiles(cc, cp, id, res, webdata);
287
288   N_VDestroy_Parallel(res);
289
290   /* Set remaining inputs to IDAMalloc. */
291
292   t0 = ZERO;
293   rtol = RTOL;
294   atol = ATOL;
295
296   /* Call IDACreate and IDAMalloc to initialize solution */
297
298   mem = IDACreate();
299   if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);
300
301   retval = IDASetRdata(mem, webdata);
302   if(check_flag(&retval, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);
303
304   retval = IDASetId(mem, id);
305   if(check_flag(&retval, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);
306
307   retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, &rtol, &atol);
308   if(check_flag(&retval, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
309
310   /* Call IDABBDPrecAlloc to initialize the band-block-diagonal preconditioner.
311      The half-bandwidths for the difference quotient evaluation are exact
312      for the system Jacobian, but only a 5-diagonal band matrix is retained. */
313
314   mudq = mldq = NSMXSUB;
315   mukeep = mlkeep = 2;
316   P_data = IDABBDPrecAlloc(mem, local_N, mudq, mldq, mukeep, mlkeep,
317                            ZERO, reslocal, NULL);
318   if(check_flag((void *)P_data, "IDABBDPrecAlloc", 0, thispe)) MPI_Abort(comm, 1);
319
320   /* Call IDABBDSpgmr to specify the IDA linear solver IDASPGMR and specify
321      the preconditioner routines supplied
322      maxl (max. Krylov subspace dim.) is set to 12.   */
```

```
323
324     maxl = 12;
325     retval = IDABBDSpgmr(mem, maxl, P_data);
326     if(check_flag(&retval, "IDABBDSpgmr", 1, thispe)) MPI_Abort(comm, 1);
327
328     /* Call IDACalcIC (with default options) to correct the initial values. */
329
330     tout = RCONST(0.001);
331     retval = IDACalcIC(mem, IDA_YA_YDP_INIT, tout);
332     if(check_flag(&retval, "IDACalcIC", 1, thispe)) MPI_Abort(comm, 1);
333
334     /* On PE 0, print heading, basic parameters, initial values. */
335
336     if (thispe == 0) PrintHeader(SystemSize, maxl,
337                                  mudq, mldq, mukeep, mlkeep,
338                                  rtol, atol);
339     PrintOutput(mem, cc, t0, webdata, comm);
340
341     /* Call IDA in tout loop, normal mode, and print selected output. */
342
343     for (iout = 1; iout <= NOUT; iout++) {
344
345       retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
346       if(check_flag(&retval, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
347
348       PrintOutput(mem, cc, tret, webdata, comm);
349
350       if (iout < 3) tout *= TMULT;
351       else          tout += TADD;
352
353     }
354
355     /* On PE 0, print final set of statistics. */
356
357     if (thispe == 0)  PrintFinalStats(mem, P_data);
358
359     /* Free memory. */
360
361     N_VDestroy_Parallel(cc);
362     N_VDestroy_Parallel(cp);
363     N_VDestroy_Parallel(id);
364
365     IDABBDPrecFree(P_data);
366
367     IDAFree(mem);
368
369     denfree(webdata->acoef);
370     N_VDestroy_Parallel(webdata->rates);
371     free(webdata);
372
373     MPI_Finalize();
374
375     return(0);
376   }
```

```
377
378 /*
379  *----------------------------------------------------------------------
380  * PRIVATE FUNCTIONS
381  *----------------------------------------------------------------------
382  */
383
384 /*
385  * InitUserData: Load problem constants in webdata (of type UserData).
386  */
387
388 static void InitUserData(UserData webdata, int thispe, int npes,
389                          MPI_Comm comm)
390 {
391   int i, j, np;
392   realtype *a1,*a2, *a3, *a4, dx2, dy2, **acoef, *bcoef, *cox, *coy;
393
394   webdata->jysub = thispe / NPEX;
395   webdata->ixsub = thispe - (webdata->jysub)*NPEX;
396   webdata->mxsub = MXSUB;
397   webdata->mysub = MYSUB;
398   webdata->npex = NPEX;
399   webdata->npey = NPEY;
400   webdata->ns = NUM_SPECIES;
401   webdata->np = NPREY;
402   webdata->dx = AX/(MX-1);
403   webdata->dy = AY/(MY-1);
404   webdata->thispe = thispe;
405   webdata->npes   = npes;
406   webdata->nsmxsub = MXSUB * NUM_SPECIES;
407   webdata->nsmxsub2 = (MXSUB+2)*NUM_SPECIES;
408   webdata->comm = comm;
409   webdata->n_local = MXSUB*MYSUB*NUM_SPECIES;
410
411   /* Set up the coefficients a and b plus others found in the equations. */
412
413   np = webdata->np;
414   dx2 = (webdata->dx)*(webdata->dx);
415   dy2 = (webdata->dy)*(webdata->dy);
416
417   acoef = webdata->acoef;
418   bcoef = webdata->bcoef;
419   cox = webdata->cox;
420   coy = webdata->coy;
421
422   for (i = 0; i < np; i++) {
423     a1 = &(acoef[i][np]);
424     a2 = &(acoef[i+np][0]);
425     a3 = &(acoef[i][0]);
426     a4 = &(acoef[i+np][np]);
427     /*  Fill in the portion of acoef in the four quadrants, row by row. */
428     for (j = 0; j < np; j++) {
429       *a1++ =  -GG;
430       *a2++ =   EE;
```

```
431         *a3++ = ZERO;
432         *a4++ = ZERO;
433       }
434 
435       /* Reset the diagonal elements of acoef to -AA. */
436       acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;
437 
438       /* Set coefficients for b and diffusion terms. */
439       bcoef[i] = BB; bcoef[i+np] = -BB;
440       cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
441       coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
442     }
443 
444   }
445 
446   /*
447    * SetInitialProfiles: Set initial conditions in cc, cp, and id.
448    * A polynomial profile is used for the prey cc values, and a constant
449    * (1.0e5) is loaded as the initial guess for the predator cc values.
450    * The id values are set to 1 for the prey and 0 for the predators.
451    * The prey cp values are set according to the given system, and
452    * the predator cp values are set to zero.
453    */
454 
455   static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
456                                  N_Vector res, UserData webdata)
457   {
458     long int ixsub, jysub, mxsub, mysub, nsmxsub, np, ix, jy, is;
459     realtype *cxy, *idxy, *cpxy, dx, dy, xx, yy, xyfactor;
460 
461     ixsub = webdata->ixsub;
462     jysub = webdata->jysub;
463     mxsub = webdata->mxsub;
464     mysub = webdata->mxsub;
465     nsmxsub = webdata->nsmxsub;
466     dx = webdata->dx;
467     dy = webdata->dy;
468     np = webdata->np;
469 
470     /* Loop over grid, load cc values and id values. */
471     for (jy = 0; jy < mysub; jy++) {
472       yy = (jy + jysub*mysub) * dy;
473       for (ix = 0; ix < mxsub; ix++) {
474         xx = (ix + ixsub*mxsub) * dx;
475         xyfactor = 16.*xx*(1. - xx)*yy*(1. - yy);
476         xyfactor *= xyfactor;
477 
478         cxy = IJ_Vptr(cc,ix,jy);
479         idxy = IJ_Vptr(id,ix,jy);
480         for (is = 0; is < NUM_SPECIES; is++) {
481           if (is < np) { cxy[is] = RCONST(10.0) + (realtype)(is+1)*xyfactor; idxy[is] = ONE; }
482           else { cxy[is] = 1.0e5; idxy[is] = ZERO; }
483         }
484       }
```

```
485   }
486
487   /* Set c' for the prey by calling the residual function with cp = 0. */
488
489   N_VConst(ZERO, cp);
490   resweb(ZERO, cc, cp, res, webdata);
491   N_VScale(-ONE, res, cp);
492
493   /* Set c' for predators to 0. */
494
495   for (jy = 0; jy < mysub; jy++) {
496     for (ix = 0; ix < mxsub; ix++) {
497       cpxy = IJ_Vptr(cp,ix,jy);
498       for (is = np; is < NUM_SPECIES; is++) cpxy[is] = ZERO;
499     }
500   }
501 }
502
503 /*
504  * Print first lines of output (problem description)
505  * and table headerr
506  */
507
508 static void PrintHeader(long int SystemSize, int maxl,
509                         long int mudq, long int mldq,
510                         long int mukeep, long int mlkeep,
511                         realtype rtol, realtype atol)
512 {
513   printf("\niwebbbd: Predator-prey DAE parallel example problem for IDA \n\n");
514   printf("Number of species ns: %d", NUM_SPECIES);
515   printf("     Mesh dimensions: %d x %d", MX, MY);
516   printf("     Total system size: %ld\n",SystemSize);
517   printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
518   printf("     Processor array: %d x %d\n", NPEX, NPEY);
519 #if defined(SUNDIALS_EXTENDED_PRECISION)
520   printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
521 #elif defined(SUNDIALS_DOUBLE_PRECISION)
522   printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
523 #else
524   printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
525 #endif
526   printf("Linear solver: IDASPGMR     Max. Krylov dimension maxl: %d\n", maxl);
527   printf("Preconditioner: band-block-diagonal (IDABBDPRE), with parameters\n");
528   printf("     mudq = %ld,  mldq = %ld,  mukeep = %ld,  mlkeep = %ld\n",
529          mudq, mldq, mukeep, mlkeep);
530   printf("CalcIC called to correct initial predator concentrations \n\n");
531   printf("------------------------------------------------------------\n");
532   printf("  t        bottom-left  top-right");
533   printf("    | nst  k      h\n");
534   printf("------------------------------------------------------------\n\n");
535 }
536
537
538 /*
```

```
 * 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++)
      printf("         %12.4Le %12.4Le   |\n",cdata[i],clast[i]);
#elif defined(SUNDIALS_DOUBLE_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++)
      printf("         %12.4le %12.4le   |\n",cdata[i],clast[i]);
#else
    printf("%8.2e %12.4e %12.4e   | %3ld  %1d %12.4e\n",
         tt, cdata[0], clast[0], nst, kused, hused);
    for (i=1;i<NUM_SPECIES;i++)
```

```
593       printf("         %12.4e %12.4e   |\n",cdata[i],clast[i]);
594 #endif
595     printf("\n");
596 
597   }
598 
599 }
600 
601 /*
602  * PrintFinalStats: Print final run data contained in iopt.
603  */
604 
605 static void PrintFinalStats(void *mem, void *P_data)
606 {
607   long int nst, nre, nreS, netf, ncfn, nni, ncfl, nli, npe, nps, nge;
608   int flag;
609 
610   flag = IDAGetNumSteps(mem, &nst);
611   check_flag(&flag, "IDAGetNumSteps", 1, 0);
612   flag = IDAGetNumResEvals(mem, &nre);
613   check_flag(&flag, "IDAGetNumResEvals", 1, 0);
614   flag = IDAGetNumErrTestFails(mem, &netf);
615   check_flag(&flag, "IDAGetNumErrTestFails", 1, 0);
616   flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
617   check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1, 0);
618   flag = IDAGetNumNonlinSolvIters(mem, &nni);
619   check_flag(&flag, "IDAGetNumNonlinSolvIters", 1, 0);
620 
621   flag = IDASpgmrGetNumConvFails(mem, &ncfl);
622   check_flag(&flag, "IDASpgmrGetNumConvFails", 1, 0);
623   flag = IDASpgmrGetNumLinIters(mem, &nli);
624   check_flag(&flag, "IDASpgmrGetNumLinIters", 1, 0);
625   flag = IDASpgmrGetNumPrecEvals(mem, &npe);
626   check_flag(&flag, "IDASpgmrGetNumPrecEvals", 1, 0);
627   flag = IDASpgmrGetNumPrecSolves(mem, &nps);
628   check_flag(&flag, "IDASpgmrGetNumPrecSolves", 1, 0);
629   flag = IDASpgmrGetNumResEvals(mem, &nreS);
630   check_flag(&flag, "IDASpgmrGetNumResEvals", 1, 0);
631 
632   flag = IDABBDPrecGetNumGfnEvals(P_data, &nge);
633   check_flag(&flag, "IDABBDPrecGetNumGfnEvals", 1, 0);
634 
635   printf("-------------------------------------------------------------\n");
636   printf("\nFinal statistics: \n\n");
637 
638   printf("Number of steps                    = %ld\n", nst);
639   printf("Number of residual evaluations     = %ld\n", nre+nreS);
640   printf("Number of nonlinear iterations     = %ld\n", nni);
641   printf("Number of error test failures      = %ld\n", netf);
642   printf("Number of nonlinear conv. failures = %ld\n\n", ncfn);
643 
644   printf("Number of linear iterations        = %ld\n", nli);
645   printf("Number of linear conv. failures    = %ld\n\n", ncfl);
646 
```

```
647   printf("Number of preconditioner setups    = %ld\n", npe);
648   printf("Number of preconditioner solves    = %ld\n", nps);
649   printf("Number of local residual evals.    = %ld\n", nge);
650 
651 }
652 
653 /*
654  * Check function return value...
655  *   opt == 0 means SUNDIALS function allocates memory so check if
656  *            returned NULL pointer
657  *   opt == 1 means SUNDIALS function returns a flag so check if
658  *            flag >= 0
659  *   opt == 2 means function allocates memory so check if returned
660  *            NULL pointer
661  */
662 
663 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
664 {
665   int *errflag;
666 
667   if (opt == 0 && flagvalue == NULL) {
668     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
669     fprintf(stderr,
670             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
671             id, funcname);
672     return(1);
673   } else if (opt == 1) {
674     /* Check if flag < 0 */
675     errflag = (int *) flagvalue;
676     if (*errflag < 0) {
677       fprintf(stderr,
678               "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
679               id, funcname, *errflag);
680       return(1);
681     }
682   } else if (opt == 2 && flagvalue == NULL) {
683     /* Check if function returned NULL pointer - no memory allocated */
684     fprintf(stderr,
685             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
686             id, funcname);
687     return(1);
688   }
689 
690   return(0);
691 }
692 
693 /*
694  *------------------------------------------------------------------------
695  * FUNCTIONS CALLED BY IDA & SUPPORTING FUNCTIONS
696  *------------------------------------------------------------------------
697  */
698 
699 /*
700  * resweb: System residual function for predator-prey system.
```

```
701   * To compute the residual function F, this routine calls:
702   * rescomm, for needed communication, and then
703   * reslocal, for computation of the residuals on this processor.
704   */
705
706  static int resweb(realtype tt,
707                    N_Vector cc, N_Vector cp, N_Vector rr,
708                    void *res_data)
709  {
710    int retval;
711    UserData webdata;
712    long int Nlocal;
713
714    webdata = (UserData) res_data;
715
716    Nlocal = webdata->n_local;
717
718    /* Call rescomm to do inter-processor communication. */
719    retval = rescomm(Nlocal, tt, cc, cp, res_data);
720
721    /* Call reslocal to calculate the local portion of residual vector. */
722    retval = reslocal(Nlocal, tt, cc, cp, rr, res_data);
723
724    return(0);
725  }
726
727  /*
728   * rescomm: Communication routine in support of resweb.
729   * This routine performs all inter-processor communication of components
730   * of the cc vector needed to calculate F, namely the components at all
731   * interior subgrid boundaries (ghost cell data).  It loads this data
732   * into a work array cext (the local portion of c, extended).
733   * The message-passing uses blocking sends, non-blocking receives,
734   * and receive-waiting, in routines BRecvPost, BSend, BRecvWait.
735   */
736
737  static int rescomm(long int Nlocal, realtype tt,
738                     N_Vector cc, N_Vector cp,
739                     void *res_data)
740  {
741
742    UserData webdata;
743    realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
744    long int thispe, ixsub, jysub, nsmxsub, nsmysub;
745    MPI_Comm comm;
746    MPI_Request request[4];
747
748    webdata = (UserData) res_data;
749    cdata = NV_DATA_P(cc);
750
751    /* Get comm, thispe, subgrid indices, data sizes, extended array cext. */
752
753    comm = webdata->comm;
754    thispe = webdata->thispe;
```

```
755
756   ixsub = webdata->ixsub;
757   jysub = webdata->jysub;
758   cext = webdata->cext;
759   nsmxsub = webdata->nsmxsub;
760   nsmysub = (webdata->ns)*(webdata->mysub);
761
762   /* Start receiving boundary data from neighboring PEs. */
763
764   BRecvPost(comm, request, thispe, ixsub, jysub, nsmxsub, nsmysub,
765             cext, buffer);
766
767   /* Send data from boundary of local grid to neighboring PEs. */
768
769   BSend(comm, thispe, ixsub, jysub, nsmxsub, nsmysub, cdata);
770
771   /* Finish receiving boundary data from neighboring PEs. */
772
773   BRecvWait(request, ixsub, jysub, nsmxsub, cext, buffer);
774
775   return(0);
776 }
777
778 /*
779  * BRecvPost: Start receiving boundary data from neighboring PEs.
780  * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
781  *     should be passed to both the BRecvPost and BRecvWait functions, and
782  *     should not be manipulated between the two calls.
783  * (2) request should have 4 entries, and is also passed in both calls.
784  */
785
786 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
787                       long int ixsub, long int jysub,
788                       long int dsizex, long int dsizey,
789                       realtype cext[], realtype buffer[])
790 {
791   long int offsetce;
792   /* Have bufleft and bufright use the same buffer. */
793   realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
794
795   /* If jysub > 0, receive data for bottom x-line of cext. */
796   if (jysub != 0)
797     MPI_Irecv(&cext[NUM_SPECIES], dsizex, PVEC_REAL_MPI_TYPE,
798               my_pe-NPEX, 0, comm, &request[0]);
799
800   /* If jysub < NPEY-1, receive data for top x-line of cext. */
801   if (jysub != NPEY-1) {
802     offsetce = NUM_SPECIES*(1 + (MYSUB+1)*(MXSUB+2));
803     MPI_Irecv(&cext[offsetce], dsizex, PVEC_REAL_MPI_TYPE,
804               my_pe+NPEX, 0, comm, &request[1]);
805   }
806
807   /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
808   if (ixsub != 0) {
```

```
809       MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
810                 my_pe-1, 0, comm, &request[2]);
811     }
812
813     /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
814     if (ixsub != NPEX-1) {
815       MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
816                 my_pe+1, 0, comm, &request[3]);
817     }
818
819   }
820
821   /*
822    * BRecvWait: Finish receiving boundary data from neighboring PEs.
823    * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
824    *     should be passed to both the BRecvPost and BRecvWait functions, and
825    *     should not be manipulated between the two calls.
826    * (2) request should have 4 entries, and is also passed in both calls.
827    */
828
829   static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
830                         long int dsizex, realtype cext[], realtype buffer[])
831   {
832     int i;
833     long int ly, dsizex2, offsetce, offsetbuf;
834     realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
835     MPI_Status status;
836
837     dsizex2 = dsizex + 2*NUM_SPECIES;
838
839     /* If jysub > 0, receive data for bottom x-line of cext. */
840     if (jysub != 0)
841       MPI_Wait(&request[0],&status);
842
843     /* If jysub < NPEY-1, receive data for top x-line of cext. */
844     if (jysub != NPEY-1)
845       MPI_Wait(&request[1],&status);
846
847     /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
848     if (ixsub != 0) {
849       MPI_Wait(&request[2],&status);
850
851       /* Copy the buffer to cext */
852       for (ly = 0; ly < MYSUB; ly++) {
853         offsetbuf = ly*NUM_SPECIES;
854         offsetce = (ly+1)*dsizex2;
855         for (i = 0; i < NUM_SPECIES; i++)
856           cext[offsetce+i] = bufleft[offsetbuf+i];
857       }
858     }
859
860     /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
861     if (ixsub != NPEX-1) {
862       MPI_Wait(&request[3],&status);
```

```
863
864       /* Copy the buffer to cext */
865       for (ly = 0; ly < MYSUB; ly++) {
866         offsetbuf = ly*NUM_SPECIES;
867         offsetce = (ly+2)*dsizex2 - NUM_SPECIES;
868         for (i = 0; i < NUM_SPECIES; i++)
869           cext[offsetce+i] = bufright[offsetbuf+i];
870       }
871     }
872   }
873
874   /*
875    * BSend: Send boundary data to neighboring PEs.
876    * This routine sends components of cc from internal subgrid boundaries
877    * to the appropriate neighbor PEs.
878    */
879
880   static void BSend(MPI_Comm comm, long int my_pe, long int ixsub, long int jysub,
881                     long int dsizex, long int dsizey, realtype cdata[])
882   {
883     int i;
884     long int ly, offsetc, offsetbuf;
885     realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];
886
887     /* If jysub > 0, send data from bottom x-line of cc. */
888
889     if (jysub != 0)
890       MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
891
892     /* If jysub < NPEY-1, send data from top x-line of cc. */
893
894     if (jysub != NPEY-1) {
895       offsetc = (MYSUB-1)*dsizex;
896       MPI_Send(&cdata[offsetc], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
897     }
898
899     /* If ixsub > 0, send data from left y-line of cc (via bufleft). */
900
901     if (ixsub != 0) {
902       for (ly = 0; ly < MYSUB; ly++) {
903         offsetbuf = ly*NUM_SPECIES;
904         offsetc = ly*dsizex;
905         for (i = 0; i < NUM_SPECIES; i++)
906           bufleft[offsetbuf+i] = cdata[offsetc+i];
907       }
908       MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
909     }
910
911     /* If ixsub < NPEX-1, send data from right y-line of cc (via bufright). */
912
913     if (ixsub != NPEX-1) {
914       for (ly = 0; ly < MYSUB; ly++) {
915         offsetbuf = ly*NUM_SPECIES;
916         offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
```

```
917         for (i = 0; i < NUM_SPECIES; i++)
918           bufright[offsetbuf+i] = cdata[offsetc+i];
919       }
920       MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
921     }
922   }
923
924   /* Define lines are for ease of readability in the following functions. */
925
926   #define mxsub      (webdata->mxsub)
927   #define mysub      (webdata->mysub)
928   #define npex       (webdata->npex)
929   #define npey       (webdata->npey)
930   #define ixsub      (webdata->ixsub)
931   #define jysub      (webdata->jysub)
932   #define nsmxsub    (webdata->nsmxsub)
933   #define nsmxsub2   (webdata->nsmxsub2)
934   #define np         (webdata->np)
935   #define dx         (webdata->dx)
936   #define dy         (webdata->dy)
937   #define cox        (webdata->cox)
938   #define coy        (webdata->coy)
939   #define rhs        (webdata->rhs)
940   #define cext       (webdata->cext)
941   #define rates      (webdata->rates)
942   #define ns         (webdata->ns)
943   #define acoef      (webdata->acoef)
944   #define bcoef      (webdata->bcoef)
945
946   /*
947    * reslocal: Compute res = F(t,cc,cp).
948    * This routine assumes that all inter-processor communication of data
949    * needed to calculate F has already been done.  Components at interior
950    * subgrid boundaries are assumed to be in the work array cext.
951    * The local portion of the cc vector is first copied into cext.
952    * The exterior Neumann boundary conditions are explicitly handled here
953    * by copying data from the first interior mesh line to the ghost cell
954    * locations in cext.  Then the reaction and diffusion terms are
955    * evaluated in terms of the cext array, and the residuals are formed.
956    * The reaction terms are saved separately in the vector webdata->rates
957    * for use by the preconditioner setup routine.
958    */
959
960   static int reslocal(long int Nlocal, realtype tt,
961                       N_Vector cc, N_Vector cp, N_Vector rr,
962                       void *res_data)
963   {
964     realtype *cdata, *ratesxy, *cpxy, *resxy,
965       xx, yy, dcyli, dcyui, dcxli, dcxui;
966     long int ix, jy, is, i, locc, ylocce, locce;
967     UserData webdata;
968
969     webdata = (UserData) res_data;
970
```

```
971   /* Get data pointers, subgrid data, array sizes, work array cext. */
972
973   cdata = NV_DATA_P(cc);
974
975   /* Copy local segment of cc vector into the working extended array cext. */
976
977   locc = 0;
978   locce = nsmxsub2 + NUM_SPECIES;
979   for (jy = 0; jy < mysub; jy++) {
980     for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
981     locc = locc + nsmxsub;
982     locce = locce + nsmxsub2;
983   }
984
985   /* To facilitate homogeneous Neumann boundary conditions, when this is
986      a boundary PE, copy data from the first interior mesh line of cc to cext. */
987
988   /* If jysub = 0, copy x-line 2 of cc to cext. */
989   if (jysub == 0)
990     { for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i]; }
991
992   /* If jysub = npey-1, copy x-line mysub-1 of cc to cext. */
993   if (jysub == npey-1) {
994     locc = (mysub-2)*nsmxsub;
995     locce = (mysub+1)*nsmxsub2 + NUM_SPECIES;
996     for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
997   }
998
999   /* If ixsub = 0, copy y-line 2 of cc to cext. */
1000  if (ixsub == 0) {
1001    for (jy = 0; jy < mysub; jy++) {
1002      locc = jy*nsmxsub + NUM_SPECIES;
1003      locce = (jy+1)*nsmxsub2;
1004      for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
1005    }
1006  }
1007
1008  /* If ixsub = npex-1, copy y-line mxsub-1 of cc to cext. */
1009  if (ixsub == npex-1) {
1010    for (jy = 0; jy < mysub; jy++) {
1011      locc  = (jy+1)*nsmxsub - 2*NUM_SPECIES;
1012      locce = (jy+2)*nsmxsub2 - NUM_SPECIES;
1013      for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
1014    }
1015  }
1016
1017  /* Loop over all grid points, setting local array rates to right-hand sides.
1018     Then set rr values appropriately for prey/predator components of F. */
1019
1020  for (jy = 0; jy < mysub; jy++) {
1021    ylocce = (jy+1)*nsmxsub2;
1022    yy     = (jy+jysub*mysub)*dy;
1023
1024    for (ix = 0; ix < mxsub; ix++) {
```

```
1025        locce = ylocce + (ix+1)*NUM_SPECIES;
1026        xx = (ix + ixsub*mxsub)*dx;
1027
1028        ratesxy = IJ_Vptr(rates,ix,jy);
1029        WebRates(xx, yy, &(cext[locce]), ratesxy, webdata);
1030
1031        resxy = IJ_Vptr(rr,ix,jy);
1032        cpxy = IJ_Vptr(cp,ix,jy);
1033
1034        for (is = 0; is < NUM_SPECIES; is++) {
1035          dcyli = cext[locce+is]          - cext[locce+is-nsmxsub2];
1036          dcyui = cext[locce+is+nsmxsub2] - cext[locce+is];
1037
1038          dcxli = cext[locce+is]             - cext[locce+is-NUM_SPECIES];
1039          dcxui = cext[locce+is+NUM_SPECIES] - cext[locce+is];
1040
1041          rhs[is] = cox[is]*(dcxui-dcxli) + coy[is]*(dcyui-dcyli) + ratesxy[is];
1042
1043          if (is < np) resxy[is] = cpxy[is] - rhs[is];
1044          else         resxy[is] =          - rhs[is];
1045
1046        }
1047      }
1048    }
1049
1050    return(0);
1051  }
1052
1053  /*
1054   * WebRates: Evaluate reaction rates at a given spatial point.
1055   * At a given (x,y), evaluate the array of ns reaction terms R.
1056   */
1057
1058  static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
1059                       UserData webdata)
1060  {
1061    int is;
1062    realtype fac;
1063
1064    for (is = 0; is < NUM_SPECIES; is++)
1065      ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);
1066
1067    fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);
1068
1069    for (is = 0; is < NUM_SPECIES; is++)
1070      ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );
1071
1072  }
1073
1074  /*
1075   * dotprod: dot product routine for realtype arrays, for use by WebRates.
1076   */
1077
1078  static realtype dotprod(long int size, realtype *x1, realtype *x2)
```

```
1079 {
1080   long int i;
1081   realtype *xx1, *xx2, temp = ZERO;
1082
1083   xx1 = x1;
1084   xx2 = x2;
1085   for (i = 0; i < size; i++)
1086     temp += (*xx1++) * (*xx2++);
1087
1088   return(temp);
1089 }
1090
```