

Example Programs for IDA v2.3.0

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

April 2005

DISCLAIMER

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

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

Contents

1	Introduction	1
2	Serial example problems	3
2.1	A dense example: <code>irobx</code>	3
2.2	A banded example: <code>iwebsb</code>	5
2.3	A Krylov example: <code>iheatsk</code>	8
3	Parallel example problems	11
3.1	A user preconditioner example: <code>iheatpk</code>	11
3.2	An IDABBDPRE preconditioner example: <code>iwebbbd</code>	12
	References	15
A	Listing of <code>irobx.c</code>	16
B	Listing of <code>iwebsb.c</code>	23
C	Listing of <code>iheatsk.c</code>	36
D	Listing of <code>iheatpk.c</code>	47
E	Listing of <code>iwebbbd.c</code>	64

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

- `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:

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

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 §5.5.3.

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 dependent variable vectors into `yy` and `yp` and set the relative 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 `rto1` and the absolute tolerance vector `avto1`. 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 $t = \text{tout}$, putting the computed value of $y(\text{tout})$ and $y'(\text{tout})$ into `yy` and `yp`, respectively, with `t = tout`. If an error occurred 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 `avto1`.

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: rto1 = 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 $1, \dots, p$ (the prey) satisfy rate equations, while species $p + 1, \dots, s$ (the predators) have infinitely fast reaction rates. The coupled PDEs for the species concentrations $c^i(x, y, t)$ are:

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

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:

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

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

and

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

The spatial domain is the unit square $0 \leq x, y \leq 1$, and the time interval is $0 \leq t \leq 1$. 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:

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

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 $\text{mu} = \text{ml} = sL = 40$.

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 $\Omega = \{0 \leq x, y \leq 1\}$ and the equations solved are:

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

The time interval is $0 \leq t \leq 10.24$, 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-Schmidt 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` Gram-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 preconditioner 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 = α` 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
-----
0.01  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
-----
0.01  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 $NPEX \times NPEY$ processor grid, with an $MXSUB \times MYSUB$ 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 $NUM_SPECIES \cdot MXSUB$. 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 | 190  1   4.1353e-02
           2.4019e+06   2.7072e+06 |
4.00e-01   2.4019e+02   2.7072e+02 | 193  1   3.3082e-01
           2.4019e+06   2.7072e+06 |
7.00e-01   2.4019e+02   2.7072e+02 | 193  1   3.3082e-01
           2.4019e+06   2.7072e+06 |
1.00e+00   2.4019e+02   2.7072e+02 | 194  1   6.6164e-01
           2.4019e+06   2.7072e+06 |
-----

Final statistics:

Number of steps                = 194
Number of residual evaluations = 899
Number of nonlinear iterations = 237
Number of error test failures  = 0
Number of nonlinear conv. failures = 0

```

Number of linear iterations	= 660
Number of linear conv. failures	= 0
Number of preconditioner setups	= 26
Number of preconditioner solves	= 899
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

```
1  /*
2  * -----
3  * $Revision: 1.16.2.4 $
4  * $Date: 2005/04/06 23:34:13 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * This simple example problem for IDA/IDAS, due to Robertson,
10 * is from chemical kinetics, and consists of the following three
11 * equations:
12 *
13 *     dy1/dt = -.04*y1 + 1.e4*y2*y3
14 *     dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*y2**2
15 *     0      = y1 + y2 + y3 - 1
16 *
17 * on the interval from t = 0.0 to t = 4.e10, with initial
18 * conditions: y1 = 1, y2 = y3 = 0.
19 *
20 * The problem is solved with IDA/IDAS using IDADENSE for the linear
21 * solver, with a user-supplied Jacobian. Output is printed at
22 * t = .4, 4, 40, ..., 4e10.
23 * -----
24 */
25
26 #include <stdio.h>
27 #include <math.h>
28 #include "sundialstypes.h"
29 #include "sundialsmath.h"
30 #include "nvector_serial.h"
31 #include "ida.h"
32 #include "idadense.h"
33
34 /* Problem Constants */
35
36 #define NEQ    3
37 #define NOUT  12
38
39 #define ZERO  RCONST(0.0);
40 #define ONE   RCONST(1.0);
41
42 /* Macro to define dense matrix elements, indexed from 1. */
43
44 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1)
45
46 /* Prototypes of functions called by IDA */
47
48 int resrob(realtype tres, N_Vector yy, N_Vector yp,
49           N_Vector resval, void *rdata);
50
51 int jacob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
52           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  PrintHeader(rtol, avtol, yy);
113
114  /* Call IDACreate and IDAMalloc to initialize IDA memory */
115
116  mem = IDACreate();
117  if(check_flag((void *)mem, "IDACreate", 0)) return(1);
118  retval = IDAMalloc(mem, resrob, t0, yy, yp, IDA_SV, rtol, avtol);
119  if(check_flag(&retval, "IDAMalloc", 1)) return(1);
120
121  /* Free avtol */
122  N_VDestroy_Serial(avtol);
123
124  /* Call IDADense and set up the linear solver. */
125
126  retval = IDADense(mem, NEQ);
127  if(check_flag(&retval, "IDADense", 1)) return(1);
128  retval = IDADenseSetJacFn(mem, jacob, NULL);
129  if(check_flag(&retval, "IDADenseSetJacFn", 1)) return(1);
130
131  /* Loop over tout values and call IDASolve. */
132
133  for (tout = t1, iout = 1; iout <= NOUT ; iout++, tout *= RCONST(10.0)) {
134      retval=IDASolve(mem, tout, &tret, yy, yp, IDA_NORMAL);
135      if(check_flag(&retval, "IDASolve", 1)) return(1);
136      PrintOutput(mem,tret,yy);
137  }
138
139  PrintFinalStats(mem);
140
141  /* Free memory */
142
143  IDAFree(mem);
144  N_VDestroy_Serial(yy);
145  N_VDestroy_Serial(yp);
146
147  return(0);
148
149  }
150
151  /*
152  *-----
153  * FUNCTIONS CALLED BY IDA
154  *-----
155  */
156
157  /*
158  * Define the system residual function.
159  */
160

```

```

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

```

```

215 static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y)
216 {
217     realtype *atval, *yval;
218
219     atval = NV_DATA_S(avtol);
220     yval = NV_DATA_S(y);
221
222     printf("\nirobx: Robertson kinetics DAE serial example problem for IDA \n");
223     printf("        Three equation chemical kinetics problem. \n\n");
224     printf("Linear solver: IDADENSE, with user-supplied Jacobian.\n");
225 #if defined(SUNDIALS_EXTENDED_PRECISION)
226     printf("Tolerance parameters: rtol = %Lg atol = %Lg %Lg %Lg \n",
227           rtol, atval[0], atval[1], atval[2]);
228     printf("Initial conditions y0 = (%Lg %Lg %Lg)\n",
229           yval[0], yval[1], yval[2]);
230 #elif defined(SUNDIALS_DOUBLE_PRECISION)
231     printf("Tolerance parameters: rtol = %lg atol = %lg %lg %lg \n",
232           rtol, atval[0], atval[1], atval[2]);
233     printf("Initial conditions y0 = (%lg %lg %lg)\n",
234           yval[0], yval[1], yval[2]);
235 #else
236     printf("Tolerance parameters: rtol = %g atol = %g %g %g \n",
237           rtol, atval[0], atval[1], atval[2]);
238     printf("Initial conditions y0 = (%g %g %g)\n",
239           yval[0], yval[1], yval[2]);
240 #endif
241     printf("Constraints and id not used.\n\n");
242     printf("-----\n");
243     printf(" t          y1          y2          y3");
244     printf(" | nst k          h\n");
245     printf("-----\n");
246
247 }
248
249 /*
250  * Print Output
251  */
252
253 static void PrintOutput(void *mem, realtype t, N_Vector y)
254 {
255     realtype *yval;
256     int retval, kused;
257     long int nst;
258     realtype hused;
259
260     yval = NV_DATA_S(y);
261
262     retval = IDAGetLastOrder(mem, &kused);
263     check_flag(&retval, "IDAGetLastOrder", 1);
264     retval = IDAGetNumSteps(mem, &nst);
265     check_flag(&retval, "IDAGetNumSteps", 1);
266     retval = IDAGetLastStep(mem, &hused);
267     check_flag(&retval, "IDAGetLastStep", 1);
268 #if defined(SUNDIALS_EXTENDED_PRECISION)

```

```

269     printf("%8.2Le %12.4Le %12.4Le %12.4Le | %3ld %1d %12.4Le\n",
270           t, yval[0], yval[1], yval[2], nst, kused, hused);
271 #elif defined(SUNDIALS_DOUBLE_PRECISION)
272     printf("%8.2le %12.4le %12.4le %12.4le | %3ld %1d %12.4le\n",
273           t, yval[0], yval[1], yval[2], nst, kused, hused);
274 #else
275     printf("%8.2e %12.4e %12.4e %12.4e | %3ld %1d %12.4e\n",
276           t, yval[0], yval[1], yval[2], nst, kused, hused);
277 #endif
278 }
279
280 /*
281  * Print final integrator statistics
282  */
283
284 static void PrintFinalStats(void *mem)
285 {
286     int retval;
287     long int nst, nni, njeD, nre, nreD, netf, ncf;
288
289     retval = IDAGetNumSteps(mem, &nst);
290     check_flag(&retval, "IDAGetNumSteps", 1);
291     retval = IDAGetNumResEvals(mem, &nre);
292     check_flag(&retval, "IDAGetNumResEvals", 1);
293     retval = IDADenseGetNumJacEvals(mem, &njeD);
294     check_flag(&retval, "IDADenseGetNumJacEvals", 1);
295     retval = IDAGetNumNonlinSolvIters(mem, &nni);
296     check_flag(&retval, "IDAGetNumNonlinSolvIters", 1);
297     retval = IDAGetNumErrTestFails(mem, &netf);
298     check_flag(&retval, "IDAGetNumErrTestFails", 1);
299     retval = IDAGetNumNonlinSolvConvFails(mem, &ncf);
300     check_flag(&retval, "IDAGetNumNonlinSolvConvFails", 1);
301     retval = IDADenseGetNumResEvals(mem, &nreD);
302     check_flag(&retval, "IDADenseGetNumResEvals", 1);
303
304     printf("\nFinal Run Statistics: \n\n");
305     printf("Number of steps                = %ld\n", nst);
306     printf("Number of residual evaluations      = %ld\n", nre+nreD);
307     printf("Number of Jacobian evaluations       = %ld\n", njeD);
308     printf("Number of nonlinear iterations       = %ld\n", nni);
309     printf("Number of error test failures        = %ld\n", netf);
310     printf("Number of nonlinear conv. failures = %ld\n", ncf);
311
312 }
313
314 /*
315  * Check function return value...
316  *   opt == 0 means SUNDIALS function allocates memory so check if
317  *   returned NULL pointer
318  *   opt == 1 means SUNDIALS function returns a flag so check if
319  *   flag >= 0
320  *   opt == 2 means function allocates memory so check if returned
321  *   NULL pointer
322  */

```

```

323
324 static int check_flag(void *flagvalue, char *funcname, int opt)
325 {
326     int *errflag;
327     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
328     if (opt == 0 && flagvalue == NULL) {
329         fprintf(stderr,
330             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
331             funcname);
332         return(1);
333     } else if (opt == 1) {
334         /* Check if flag < 0 */
335         errflag = (int *) flagvalue;
336         if (*errflag < 0) {
337             fprintf(stderr,
338                 "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
339                 funcname, *errflag);
340             return(1);
341         }
342     } else if (opt == 2 && flagvalue == NULL) {
343         /* Check if function returned NULL pointer - no memory allocated */
344         fprintf(stderr,
345             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
346             funcname);
347         return(1);
348     }
349
350     return(0);
351 }

```

B Listing of iwebsb.c

```

1  /*
2  * -----
3  * $Revision: 1.18.2.2 $
4  * $Date: 2005/04/04 22:36:47 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                 Radu Serban @ LLNL
8  * -----
9  * Example program for IDA: Food web problem.
10 *
11 * This example program (serial version) uses the IDABAND linear
12 * solver, and IDACalcIC for initial condition calculation.
13 *
14 * The mathematical problem solved in this example is a DAE system
15 * that arises from a system of partial differential equations after
16 * spatial discretization. The PDE system is a food web population
17 * model, with predator-prey interaction and diffusion on the unit
18 * square in two dimensions. The dependent variable vector is:
19 *
20 *      1  2      ns
21 *   c = (c , c , ..., c ) , ns = 2 * np
22 *
23 * and the PDE's are as follows:
24 *
25 *      i      i      i
26 *   dc /dt = d(i)*(c  + c ) + R (x,y,c)   (i = 1,...,np)
27 *                xx    yy      i
28 *
29 *      i      i
30 *   0 = d(i)*(c  + c ) + R (x,y,c)   (i = np+1,...,ns)
31 *                xx    yy      i
32 *
33 *   where the reaction terms R are:
34 *
35 *      i      ns      j
36 *   R (x,y,c) = c  * (b(i) + sum a(i,j)*c )
37 *      i      j=1
38 *
39 * The number of species is ns = 2 * np, with the first np being
40 * prey and the last np being predators. The coefficients a(i,j),
41 * b(i), d(i) are:
42 *
43 * a(i,i) = -AA   (all i)
44 * a(i,j) = -GG   (i <= np , j > np)
45 * a(i,j) = EE   (i > np, j <= np)
46 * all other a(i,j) = 0
47 * b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i <= np)
48 * b(i) = -BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i > np)
49 * d(i) = DPREY   (i <= np)
50 * d(i) = DPRED   (i > np)
51 *
52 * The various scalar parameters required are set using '#define'

```

```

53 * statements or directly in routine InitUserData. In this program,
54 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
55 * normal derivative = 0.
56 *
57 * A polynomial in x and y is used to set the initial values of the
58 * first np variables (the prey variables) at each x,y location,
59 * while initial values for the remaining (predator) variables are
60 * set to a flat value, which is corrected by IDACalcIC.
61 *
62 * The PDEs are discretized by central differencing on a MX by MY
63 * mesh.
64 *
65 * The DAE system is solved by IDA using the IDABAND linear solver.
66 * Output is printed at t = 0, .001, .01, .1, .4, .7, 1.
67 * -----
68 * References:
69 * [1] Peter N. Brown and Alan C. Hindmarsh,
70 *     Reduced Storage Matrix Methods in Stiff ODE systems, Journal
71 *     of Applied Mathematics and Computation, Vol. 31 (May 1989),
72 *     pp. 40-91.
73 *
74 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
75 *     Using Krylov Methods in the Solution of Large-Scale
76 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
77 *     (1994), pp. 1467-1488.
78 *
79 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
80 *     Consistent Initial Condition Calculation for Differential-
81 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
82 *     pp. 1495-1512.
83 * -----
84 */
85
86 #include <stdio.h>
87 #include <stdlib.h>
88 #include <math.h>
89 #include "sundialstypes.h" /* Definitions of reatype and booleantype */
90 #include "ida.h"           /* Main header file */
91 #include "idaband.h"       /* Use IDABAND linear solver */
92 #include "nvector_serial.h" /* Definitions of type N_Vector, macro NV_DATA_S */
93 #include "smalldense.h"    /* Contains definitions for denalloc routine */
94
95 /* Problem Constants. */
96
97 #define NPREY      1          /* No. of prey (= no. of predators). */
98 #define NUM_SPECIES 2*NPREY
99
100 #define PI         RCONST(3.1415926535898)
101 #define FOURPI    (RCONST(4.0)*PI)
102
103 #define MX         20        /* MX = number of x mesh points */
104 #define MY         20        /* MY = number of y mesh points */
105 #define NSMX      (NUM_SPECIES * MX)
106 #define NEQ       (NUM_SPECIES*MX*MY)

```

```

107 #define AA          RCONST(1.0)    /* Coefficient in above eqns. for a */
108 #define EE          RCONST(10000.) /* Coefficient in above eqns. for a */
109 #define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a */
110 #define BB          RCONST(1.0)    /* Coefficient in above eqns. for b */
111 #define DPREY       RCONST(1.0)    /* Coefficient in above eqns. for d */
112 #define DPRED       RCONST(0.05)   /* Coefficient in above eqns. for d */
113 #define ALPHA       RCONST(50.)    /* Coefficient alpha in above eqns. */
114 #define BETA        RCONST(1000.)  /* Coefficient beta in above eqns. */
115 #define AX          RCONST(1.0)    /* Total range of x variable      */
116 #define AY          RCONST(1.0)    /* Total range of y variable      */
117 #define RTOL        RCONST(1.e-5)  /* Relative tolerance              */
118 #define ATOL        RCONST(1.e-5)  /* Absolute tolerance              */
119 #define NOUT        6               /* Number of output times          */
120 #define TMULT       RCONST(10.0)   /* Multiplier for tout values      */
121 #define TADD        RCONST(0.3)    /* Increment for tout values       */
122 #define ZERO        RCONST(0.)
123 #define ONE         RCONST(1.0)
124
125 /*
126  * User-defined vector and accessor macro: IJ_Vptr.
127  * IJ_Vptr is defined in order to express the underlying 3-D structure of
128  * the dependent variable vector from its underlying 1-D storage (an N_Vector).
129  * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
130  * species index is = 0, x-index ix = i, and y-index jy = j.
131  */
132
133 #define IJ_Vptr(vv,i,j) (&NV_Ith_S(vv, (i)*NUM_SPECIES + (j)*NSMX))
134
135 /* Type: UserData. Contains problem constants, etc. */
136
137 typedef struct {
138     long int Neq, ns, np, mx, my;
139     realtype dx, dy, **acoef;
140     realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef [NUM_SPECIES];
141     N_Vector rates;
142 } *UserData;
143
144 /* Prototypes for functions called by the IDA Solver. */
145
146 static int resweb(realtype time, N_Vector cc, N_Vector cp, N_Vector resval,
147                 void *rdata);
148
149 /* Prototypes for private Helper Functions. */
150
151 static void InitUserData(UserData webdata);
152 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
153                               UserData webdata);
154 static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol);
155 static void PrintOutput(void *mem, N_Vector c, realtype t);
156 static void PrintFinalStats(void *mem);
157 static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate, UserData webdata);
158 static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
159                     UserData webdata);
160 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, t0, cc, cp, 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     return(0);
270 }
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         }
323     }

```

```

323     }
324
325     return(0);
326
327 }
328
329 /*
330 *-----
331 * PRIVATE FUNCTIONS
332 *-----
333 */
334
335 /*
336 * InitUserData: Load problem constants in webdata (of type UserData).
337 */
338
339 static void InitUserData(UserData webdata)
340 {
341     int i, j, np;
342     realtype *a1,*a2, *a3, *a4, dx2, dy2;
343
344     webdata->mx = MX;
345     webdata->my = MY;
346     webdata->ns = NUM_SPECIES;
347     webdata->np = NPREY;
348     webdata->dx = AX/(MX-1);
349     webdata->dy = AY/(MY-1);
350     webdata->Neq= NEQ;
351
352     /* Set up the coefficients a and b, and others found in the equations. */
353     np = webdata->np;
354     dx2 = (webdata->dx)*(webdata->dx); dy2 = (webdata->dy)*(webdata->dy);
355
356     for (i = 0; i < np; i++) {
357         a1 = &(acoef[i][np]);
358         a2 = &(acoef[i+np][0]);
359         a3 = &(acoef[i][0]);
360         a4 = &(acoef[i+np][np]);
361         /* Fill in the portion of acoef in the four quadrants, row by row. */
362         for (j = 0; j < np; j++) {
363             *a1++ = -GG;
364             *a2++ = EE;
365             *a3++ = ZERO;
366             *a4++ = ZERO;
367         }
368
369         /* Reset the diagonal elements of acoef to -AA. */
370         acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;
371
372         /* Set coefficients for b and diffusion terms. */
373         bcoef[i] = BB; bcoef[i+np] = -BB;
374         cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
375         coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
376     }

```

```

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;

```

```

431     for (jx = 0; jx < MX; jx++) {
432         loc = yloc + NUM_SPECIES * jx;
433         for (is = np; is < NUM_SPECIES; is++) {
434             cpv[loc+is] = ZERO;
435         }
436     }
437 }
438 }
439
440 /*
441  * Print first lines of output (problem description)
442  */
443
444 static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol)
445 {
446     printf("\niwebsb: Predator-prey DAE serial example problem for IDA \n\n");
447     printf("Number of species ns: %d", NUM_SPECIES);
448     printf("    Mesh dimensions: %d x %d", MX, MY);
449     printf("    System size: %d\n", NEQ);
450 #if defined(SUNDIALS_EXTENDED_PRECISION)
451     printf("Tolerance parameters:  rtol = %Lg  atol = %Lg\n", rtol, atol);
452 #elif defined(SUNDIALS_DOUBLE_PRECISION)
453     printf("Tolerance parameters:  rtol = %lg  atol = %lg\n", rtol, atol);
454 #else
455     printf("Tolerance parameters:  rtol = %g  atol = %g\n", rtol, atol);
456 #endif
457     printf("Linear solver: IDABAND,  Band parameters mu = %ld, ml = %ld\n",mu,ml);
458     printf("CalcIC called to correct initial predator concentrations.\n\n");
459     printf("-----\n");
460     printf(" t          bottom-left top-right");
461     printf(" | nst k          h\n");
462     printf("-----\n\n");
463
464 }
465
466 /*
467  * PrintOutput: Print output values at output time t = tt.
468  * Selected run statistics are printed.  Then values of the concentrations
469  * are printed for the bottom left and top right grid points only.
470  */
471
472 static void PrintOutput(void *mem, N_Vector c, realtype t)
473 {
474     int i, kused, flag;
475     long int nst;
476     realtype *c_bl, *c_tr, hused;
477
478     flag = IDAGetLastOrder(mem, &kused);
479     check_flag(&flag, "IDAGetLastOrder", 1);
480     flag = IDAGetNumSteps(mem, &nst);
481     check_flag(&flag, "IDAGetNumSteps", 1);
482     flag = IDAGetLastStep(mem, &hused);
483     check_flag(&flag, "IDAGetLastStep", 1);
484

```

```

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

```
1  /*
2  * -----
3  * $Revision: 1.16.2.4 $
4  * $Date: 2005/04/06 23:34:13 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem for IDA/IDAS: 2D heat equation, serial, GMRES.
10 *
11 * This example solves a discretized 2D heat equation problem.
12 * This version uses the Krylov solver IDASpgmr.
13 *
14 * The DAE system solved is a spatial discretization of the PDE
15 *     du/dt = d^2u/dx^2 + d^2u/dy^2
16 * on the unit square. The boundary condition is u = 0 on all edges.
17 * Initial conditions are given by u = 16 x (1 - x) y (1 - y). The
18 * PDE is treated with central differences on a uniform M x M grid.
19 * The values of u at the interior points satisfy ODEs, and
20 * equations u = 0 at the boundaries are appended, to form a DAE
21 * system of size N = M^2. Here M = 10.
22 *
23 * The system is solved with IDA/IDAS using the Krylov linear solver
24 * IDASPGMR. The preconditioner uses the diagonal elements of the
25 * Jacobian only. Routines for preconditioning, required by
26 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
27 * for all components. Output is taken at t = 0, .01, .02, .04,
28 * ..., 10.24. Two cases are run -- with the Gram-Schmidt type
29 * being Modified in the first case, and Classical in the second.
30 * The second run uses IDAReInit and IDAReInitSpgmr.
31 * -----
32 */
33
34 #include <stdio.h>
35 #include <stdlib.h>
36 #include <math.h>
37 #include "sundialstypes.h"
38 #include "nvector_serial.h"
39 #include "ida.h"
40 #include "idaspgmr.h"
41
42 /* Problem Constants */
43
44 #define NOUT 11
45 #define MGRID 10
46 #define NEQ MGRID*MGRID
47 #define ZERO RCONST(0.0)
48 #define ONE RCONST(1.0)
49 #define TWO RCONST(2.0)
50 #define FOUR RCONST(4.0)
51
52 /* 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, ncf, 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  N_VDestroy_Serial(constraints);
156
157  ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
158  if(check_flag(&ier, "IDAMalloc", 1)) return(1);
159
160  /* Call IDASpgmr to specify the linear solver. */

```

```

161
162   ier = IDASpgmr(mem, 0);
163   if(check_flag(&ier, "IDASpgmr", 1)) return(1);
164
165   ier = IDASpgmrSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
166   if(check_flag(&ier, "IDASpgmrSetPreconditioner", 1)) return(1);
167
168   /* Print output heading. */
169   PrintHeader(rtol, atol);
170
171   /*
172    * -----
173    * CASE I
174    * -----
175    */
176
177   /* Print case number, output table heading, and initial line of table. */
178
179   printf("\n\nCase 1: gsytpe = MODIFIED_GS\n");
180   printf("\n  Output Summary (umax = max-norm of solution) \n\n");
181   printf("  time      umax      k  nst  nni  nje  nre  nreS      h      npe  nps\n" );
182   printf("-----\n");
183
184   /* Loop over output times, call IDASolve, and print results. */
185
186   for (tout = t1, iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
187     ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
188     if(check_flag(&ier, "IDASolve", 1)) return(1);
189     PrintOutput(mem, tret, uu);
190   }
191
192   /* Print remaining counters. */
193
194   ier = IDAGetNumErrTestFails(mem, &netf);
195   check_flag(&ier, "IDAGetNumErrTestFails", 1);
196
197   ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
198   check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
199
200   ier = IDASpgmrGetNumConvFails(mem, &ncfl);
201   check_flag(&ier, "IDASpgmrGetNumConvFails", 1);
202
203   printf("\nError test failures          = %ld\n", netf);
204   printf("Nonlinear convergence failures = %ld\n", ncfn);
205   printf("Linear convergence failures     = %ld\n", ncfl);
206
207   /*
208    * -----
209    * CASE II
210    * -----
211    */
212
213   /* Re-initialize uu, up. */
214

```

```

215 SetInitialProfile(data, uu, up, res);
216
217 /* Re-initialize IDA and IDASPGMR */
218
219 ier = IDAReInit(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
220 if(check_flag(&ier, "IDAReInit", 1)) return(1);
221
222 ier = IDASpgmrSetGSType(mem, CLASSICAL_GS);
223 if(check_flag(&ier, "IDASpgmrSetGSType",1)) return(1);
224
225 /* Print case number, output table heading, and initial line of table. */
226
227 printf("\n\nCase 2: gstype = CLASSICAL_GS\n");
228 printf("\n Output Summary (umax = max-norm of solution) \n\n");
229 printf(" time      umax      k nst nni nje nre nreS      h      npe nps\n" );
230 printf("-----\n");
231
232 /* Loop over output times, call IDASolve, and print results. */
233
234 for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
235     ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
236     if(check_flag(&ier, "IDASolve", 1)) return(1);
237     PrintOutput(mem, tret, uu);
238 }
239
240 /* Print remaining counters. */
241
242 ier = IDAGetNumErrTestFails(mem, &netf);
243 check_flag(&ier, "IDAGetNumErrTestFails", 1);
244
245 ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
246 check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
247
248 ier = IDASpgmrGetNumConvFails(mem, &ncfl);
249 check_flag(&ier, "IDASpgmrGetNumConvFails", 1);
250
251 printf("\nError test failures          = %ld\n", netf);
252 printf("Nonlinear convergence failures = %ld\n", ncfn);
253 printf("Linear convergence failures     = %ld\n", ncfl);
254
255 /* Free Memory */
256
257 IDAFree(mem);
258
259 N_VDestroy_Serial(uu);
260 N_VDestroy_Serial(up);
261 N_VDestroy_Serial(res);
262
263 N_VDestroy_Serial(data->pp);
264 free(data);
265
266 return(0);
267 }
268

```

```

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

```

```

323 * matrix P approximating the system Jacobian matrix
324 *           J = dF/du + cj*dF/du'
325 * (where the DAE system is F(t,u,u') = 0), and solve the linear
326 * systems P z = r. This is done in this case by keeping only
327 * the diagonal elements of the J matrix above, storing them as
328 * inverses in a vector pp, when computed in PsetupHeat, for
329 * subsequent use in PsolveHeat.
330 *
331 * In this instance, only cj and data (user data structure, with
332 * pp etc.) are used from the PsetupHeat argument list.
333 */
334
335 int PsetupHeat(realtype tt,
336               N_Vector uu, N_Vector up, N_Vector rr,
337               realtype c_j, void *prec_data,
338               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
339 {
340
341     long int i, j, offset, loc, mm;
342     realtype *ppv, pelinv;
343     UserData data;
344
345     data = (UserData) prec_data;
346     ppv = NV_DATA_S(data->pp);
347     mm = data->mm;
348
349     /* Initialize the entire vector to 1., then set the interior points to the
350        correct value for preconditioning. */
351     N_VConst(ONE,data->pp);
352
353     /* Compute the inverse of the preconditioner diagonal elements. */
354     pelinv = ONE/(c_j + FOUR*data->coeff);
355
356     for (j = 1; j < mm-1; j++) {
357         offset = mm * j;
358         for (i = 1; i < mm-1; i++) {
359             loc = offset + i;
360             ppv[loc] = pelinv;
361         }
362     }
363
364     return(0);
365 }
366
367 /*
368 * PsolveHeat: solve preconditioner linear system.
369 * This routine multiplies the input vector rvec by the vector pp
370 * containing the inverse diagonal Jacobian elements (previously
371 * computed in PrecondHeateq), returning the result in zvec.
372 */
373
374 int PsolveHeat(realtype tt,
375               N_Vector uu, N_Vector up, N_Vector rr,
376               N_Vector rvec, N_Vector zvec,

```

```

377         reatype c_j, reatype delta, void *prec_data,
378         N_Vector tmp)
379     {
380         UserData data;
381         data = (UserData) prec_data;
382         N_VProd(data->pp, rvec, zvec);
383         return(0);
384     }
385
386     /*
387     *-----
388     * PRIVATE FUNCTIONS
389     *-----
390     */
391
392     /*
393     * SetInitialProfile: routine to initialize u and up vectors.
394     */
395
396     static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
397                                N_Vector res)
398     {
399         long int mm, mm1, i, j, offset, loc;
400         reatype xfact, yfact, *udata, *update;
401
402         mm = data->mm;
403
404         udata = NV_DATA_S(uu);
405         update = NV_DATA_S(up);
406
407         /* Initialize uu on all grid points. */
408         mm1 = mm - 1;
409         for (j = 0; j < mm; j++) {
410             yfact = data->dx * j;
411             offset = mm*j;
412             for (i = 0; i < mm; i++) {
413                 xfact = data->dx * i;
414                 loc = offset + i;
415                 udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
416             }
417         }
418
419         /* Initialize up vector to 0. */
420         N_VConst(ZERO, up);
421
422         /* resHeat sets res to negative of ODE RHS values at interior points. */
423         resHeat(ZERO, uu, up, res, data);
424
425         /* Copy -res into up to get correct interior initial up values. */
426         N_VScale(-ONE, res, up);
427
428         /* Set up at boundary points to zero. */
429         for (j = 0; j < mm; j++) {
430             offset = mm*j;

```

```

431     for (i = 0; i < mm; i++) {
432         loc = offset + i;
433         if (j == 0 || j == mm1 || i == 0 || i == mm1 ) updata[loc] = ZERO;
434     }
435 }
436
437 return(0);
438 }
439
440 /*
441  * Print first lines of output (problem description)
442  */
443
444 static void PrintHeader(realtype rtol, realtype atol)
445 {
446     printf("\niheatsk: Heat equation, serial example problem for IDA \n");
447     printf("        Discretized heat equation on 2D unit square. \n");
448     printf("        Zero boundary conditions,");
449     printf(" polynomial initial conditions.\n");
450     printf("        Mesh dimensions: %d x %d", MGRID, MGRID);
451     printf("        Total system size: %d\n\n", NEQ);
452 #if defined(SUNDIALS_EXTENDED_PRECISION)
453     printf("Tolerance parameters:  rtol = %Lg  atol = %Lg\n", rtol, atol);
454 #elif defined(SUNDIALS_DOUBLE_PRECISION)
455     printf("Tolerance parameters:  rtol = %lg  atol = %lg\n", rtol, atol);
456 #else
457     printf("Tolerance parameters:  rtol = %g  atol = %g\n", rtol, atol);
458 #endif
459     printf("Constraints set to force all solution components >= 0. \n");
460     printf("Linear solver: IDASPGMR, preconditioner using diagonal elements. \n");
461 }
462
463 /*
464  * PrintOutput: print max norm of solution and current solver statistics
465  */
466
467 static void PrintOutput(void *mem, realtype t, N_Vector uu)
468 {
469     realtype hused, umax;
470     long int nst, nni, nje, nre, nreS, nli, npe, nps;
471     int kused, ier;
472
473     umax = N_VMaxNorm(uu);
474
475     ier = IDAGetLastOrder(mem, &kused);
476     check_flag(&ier, "IDAGetLastOrder", 1);
477     ier = IDAGetNumSteps(mem, &nst);
478     check_flag(&ier, "IDAGetNumSteps", 1);
479     ier = IDAGetNumNonlinSolvIters(mem, &nni);
480     check_flag(&ier, "IDAGetNumNonlinSolvIters", 1);
481     ier = IDAGetNumResEvals(mem, &nre);
482     check_flag(&ier, "IDAGetNumResEvals", 1);
483     ier = IDAGetLastStep(mem, &hused);
484     check_flag(&ier, "IDAGetLastStep", 1);

```

```

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

```

```
539     fprintf(stderr,  
540             "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",  
541             funcname);  
542     return(1);  
543 }  
544  
545 return(0);  
546 }
```

D Listing of iheatpk.c

```
1  /*
2  * -----
3  * $Revision: 1.15.2.3 $
4  * $Date: 2005/04/06 23:34:20 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem for IDA: 2D heat equation, parallel, GMRES.
10 *
11 * This example solves a discretized 2D heat equation problem.
12 * This version uses the Krylov solver IDASpgmr.
13 *
14 * The DAE system solved is a spatial discretization of the PDE
15 *      du/dt = d^2u/dx^2 + d^2u/dy^2
16 * on the unit square. The boundary condition is u = 0 on all edges.
17 * Initial conditions are given by u = 16 x (1 - x) y (1 - y).
18 * The PDE is treated with central differences on a uniform MX x MY
19 * grid. The values of u at the interior points satisfy ODEs, and
20 * equations u = 0 at the boundaries are appended, to form a DAE
21 * system of size N = MX * MY. Here MX = MY = 10.
22 *
23 * The system is actually implemented on submeshes, processor by
24 * processor, with an MXSUB by MYSUB mesh on each of NPEX * NPEY
25 * processors.
26 *
27 * The system is solved with IDA using the Krylov linear solver
28 * IDASPGMR. The preconditioner uses the diagonal elements of the
29 * Jacobian only. Routines for preconditioning, required by
30 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
31 * for all components. Local error testing on the boundary values
32 * is suppressed. Output is taken at t = 0, .01, .02, .04,
33 * ..., 10.24.
34 * -----
35 */
36
37 #include <stdio.h>
38 #include <stdlib.h>
39 #include <math.h>
40 #include "sundialstypes.h"
41 #include "sundialsmath.h"
42 #include "nvector_parallel.h"
43 #include "ida.h"
44 #include "idaspgmr.h"
45 #include "iterative.h"
46 #include "mpi.h"
47
48 #define ZERO  RCONST(0.0)
49 #define ONE   RCONST(1.0)
50 #define TWO   RCONST(2.0)
51
52 #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

```

```

107 /* Private function to check function return values */
108
109 static int InitUserData(int thispe, MPI_Comm comm, UserData data);
110
111 static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
112                             N_Vector res, UserData data);
113
114 static void PrintHeader(long int Neq, realtype rtol, realtype atol);
115
116 static void PrintOutput(int id, void *mem, realtype t, N_Vector uu);
117
118 static void PrintFinalStats(void *mem);
119
120 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
121
122 /*
123  *-----
124  * MAIN PROGRAM
125  *-----
126  */
127
128 int main(int argc, char *argv[])
129 {
130     MPI_Comm comm;
131     void *mem;
132     UserData data;
133     int iout, thispe, ier, npes;
134     long int Neq, local_N;
135     realtype rtol, atol, t0, t1, tout, tret;
136     N_Vector uu, up, constraints, id, res;
137
138     mem = NULL;
139     data = NULL;
140     uu = up = constraints = id = res = NULL;
141
142     /* Get processor number and total number of pe's. */
143
144     MPI_Init(&argc, &argv);
145     comm = MPI_COMM_WORLD;
146     MPI_Comm_size(comm, &npes);
147     MPI_Comm_rank(comm, &thispe);
148
149     if (npes != NPEX*NPEY) {
150         if (thispe == 0)
151             fprintf(stderr,
152                 "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n",
153                 npes, NPEX*NPEY);
154         MPI_Finalize();
155         return(1);
156     }
157
158     /* Set local length local_N and global length Neq. */
159
160     local_N = MXSUB*MYSUB;

```

```

161 Neq      = MX * MY;
162
163 /* Allocate and initialize the data structure and N-vectors. */
164
165 data = (UserData) malloc(sizeof *data);
166 data->pp = NULL;
167 if(check_flag((void *)data, "malloc", 2, thispe))
168     MPI_Abort(comm, 1);
169
170 uu = N_VNew_Parallel(comm, local_N, Neq);
171 if(check_flag((void *)uu, "N_VNew_Parallel", 0, thispe))
172     MPI_Abort(comm, 1);
173
174 up = N_VNew_Parallel(comm, local_N, Neq);
175 if(check_flag((void *)up, "N_VNew_Parallel", 0, thispe))
176     MPI_Abort(comm, 1);
177
178 res = N_VNew_Parallel(comm, local_N, Neq);
179 if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe))
180     MPI_Abort(comm, 1);
181
182 constraints = N_VNew_Parallel(comm, local_N, Neq);
183 if(check_flag((void *)constraints, "N_VNew_Parallel", 0, thispe))
184     MPI_Abort(comm, 1);
185
186 id = N_VNew_Parallel(comm, local_N, Neq);
187 if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe))
188     MPI_Abort(comm, 1);
189
190 /* An N-vector to hold preconditioner. */
191 data->pp = N_VNew_Parallel(comm, local_N, Neq);
192 if(check_flag((void *)data->pp, "N_VNew_Parallel", 0, thispe))
193     MPI_Abort(comm, 1);
194
195 InitUserData(thispe, comm, data);
196
197 /* Initialize the uu, up, id, and res profiles. */
198
199 SetInitialProfile(uu, up, id, res, data);
200
201 /* Set constraints to all 1's for nonnegative solution values. */
202
203 N_VConst(ONE, constraints);
204
205 t0 = ZERO; t1 = RCONST(0.01);
206
207 /* Scalar relative and absolute tolerance. */
208
209 rtol = ZERO;
210 atol = RCONST(1.0e-3);
211
212 /* Call IDACreate and IDAMalloc to initialize solution. */
213
214 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  N_VDestroy_Parallel(constraints);
229
230  ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
231  if(check_flag(&ier, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
232
233  /* Call IDASpgmr to specify the linear solver. */
234
235  ier = IDASpgmr(mem, 0);
236  if(check_flag(&ier, "IDASpgmr", 1, thispe)) MPI_Abort(comm, 1);
237
238  ier = IDASpgmrSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
239  if(check_flag(&ier, "IDASpgmrSetPreconditioner", 1, thispe)) MPI_Abort(comm, 1);
240
241  /* Print output heading (on processor 0 only) and intial solution */
242
243  if (thispe == 0) PrintHeader(Neq, rtol, atol);
244  PrintOutput(thispe, mem, t0, uu);
245
246  /* Loop over tout, call IDASolve, print output. */
247
248  for (tout = t1, iout = 1; iout <= NOUT; iout++, tout *= TWO) {
249
250      ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
251      if(check_flag(&ier, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
252
253      PrintOutput(thispe, mem, tret, uu);
254
255  }
256
257  /* Print remaining counters. */
258
259  if (thispe == 0) PrintFinalStats(mem);
260
261  /* Free memory */
262
263  IDAFree(mem);
264
265  N_VDestroy_Parallel(id);
266  N_VDestroy_Parallel(res);
267  N_VDestroy_Parallel(up);
268  N_VDestroy_Parallel(uu);

```

```

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

```

```

323 *           J = dF/du + cj*dF/du'
324 * (where the DAE system is F(t,u,u') = 0), and solve the linear
325 * systems P z = r. This is done in this case by keeping only
326 * the diagonal elements of the J matrix above, storing them as
327 * inverses in a vector pp, when computed in PsetupHeat, for
328 * subsequent use in PsolveHeat.
329 *
330 * In this instance, only cj and data (user data structure, with
331 * pp etc.) are used from the PsetupHeat argument list.
332 *
333 */
334
335 int PsetupHeat(realtype tt,
336               N_Vector yy, N_Vector yp, N_Vector rr,
337               realtype c_j, void *prec_data,
338               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
339 {
340     realtype *ppv, pelinv;
341     long int lx, ly, ixbegin, ixend, jybegin, jyend, locu, mxsub, mysub;
342     long int ixsub, jysub, npex, npey;
343     UserData data;
344
345     data = (UserData) prec_data;
346
347     ppv = NV_DATA_P(data->pp);
348     ixsub = data->ixsub;
349     jysub = data->jysub;
350     mxsub = data->mxsub;
351     mysub = data->mysub;
352     npex = data->npex;
353     npey = data->npey;
354
355     /* Initially set all pp elements to one. */
356     N_VConst(ONE, data->pp);
357
358     /* Prepare to loop over subgrid. */
359     ixbegin = 0;
360     ixend = mxsub-1;
361     jybegin = 0;
362     jyend = mysub-1;
363     if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
364     if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
365     pelinv = ONE/(c_j + data->coeffxy);
366
367     /* Load the inverse of the preconditioner diagonal elements
368        in loop over all the local subgrid. */
369
370     for (ly = jybegin; ly <=jyend; ly++) {
371         for (lx = ixbegin; lx <= ixend; lx++) {
372             locu = lx + ly*mxsub;
373             ppv[locu] = pelinv;
374         }
375     }
376

```

```

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

```

```

431     uext = data->uext;
432
433     /* Start receiving boundary data from neighboring PEs. */
434     BRecvPost(comm, request, thispe, ixsub, jysub, mxsub, mysub, uext, buffer);
435
436     /* Send data from boundary of local grid to neighboring PEs. */
437     BSend(comm, thispe, ixsub, jysub, mxsub, mysub, uarray);
438
439     /* Finish receiving boundary data from neighboring PEs. */
440     BRecvWait(request, ixsub, jysub, mxsub, uext, buffer);
441
442     return(0);
443
444 }
445
446 /*
447  * reslocal routine. Compute res = F(t, uu, up). This routine assumes
448  * that all inter-processor communication of data needed to calculate F
449  * has already been done, and that this data is in the work array uext.
450  */
451
452 static int reslocal(realtype tt,
453                   N_Vector uu, N_Vector up, N_Vector rr,
454                   void *res_data)
455 {
456     realtype *uext, *uuv, *upv, *resv;
457     realtype termx, termy, termctr;
458     long int lx, ly, offsetu, offsetue, locu, locue;
459     long int ixsub, jysub, mxsub, mxsub2, mysub, npex, npey;
460     long int ixbegin, ixend, jybegin, jyend;
461     UserData data;
462
463     /* Get subgrid indices, array sizes, extended work array uext. */
464
465     data = (UserData) res_data;
466     uext = data->uext;
467     uuv = NV_DATA_P(uu);
468     upv = NV_DATA_P(up);
469     resv = NV_DATA_P(rr);
470     ixsub = data->ixsub; jysub = data->jysub;
471     mxsub = data->mxsub; mxsub2 = data->mxsub + 2;
472     mysub = data->mysub; npex = data->npex; npey = data->npey;
473
474     /* Initialize all elements of rr to uu. This sets the boundary
475      elements simply without indexing hassles. */
476
477     N_VScale(ONE, uu, rr);
478
479     /* Copy local segment of u vector into the working extended array uext.
480      This completes uext prior to the computation of the rr vector. */
481
482     offsetu = 0;
483     offsetue = mxsub2 + 1;
484     for (ly = 0; ly < mysub; ly++) {

```

```

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

```

```

539
540 if (ixsub != 0) {
541     for (ly = 0; ly < MYSUB; ly++) {
542         offsetu = ly*dsizex;
543         bufleft[ly] = uarray[offsetu];
544     }
545     MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, thispe-1, 0, comm);
546 }
547
548 /* If ixsub < NPEX-1, send data from right y-line of u (via bufright). */
549
550 if (ixsub != NPEX-1) {
551     for (ly = 0; ly < MYSUB; ly++) {
552         offsetu = ly*MXSUB + (MXSUB-1);
553         bufright[ly] = uarray[offsetu];
554     }
555     MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, thispe+1, 0, comm);
556 }
557
558 return(0);
559
560 }
561
562 /*
563  * Routine to start receiving boundary data from neighboring PEs.
564  * Notes:
565  * 1) buffer should be able to hold 2*MYSUB realtype entries, should be
566  *    passed to both the BRecvPost and BRecvWait functions, and should not
567  *    be manipulated between the two calls.
568  * 2) request should have 4 entries, and should be passed in
569  *    both calls also.
570  */
571
572 static int BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
573                    long int ixsub, long int jsub,
574                    long int dsizex, long int dsizey,
575                    realtype uext[], realtype buffer[])
576 {
577     long int offsetue;
578     /* Have bufleft and bufright use the same buffer. */
579     realtype *bufleft = buffer, *bufright = buffer+MYSUB;
580
581     /* If jsub > 0, receive data for bottom x-line of uext. */
582     if (jsub != 0)
583         MPI_Irecv(&uext[1], dsizex, PVEC_REAL_MPI_TYPE,
584                 thispe-NPEX, 0, comm, &request[0]);
585
586     /* If jsub < NPEY-1, receive data for top x-line of uext. */
587     if (jsub != NPEY-1) {
588         offsetue = (1 + (MYSUB+1)*(MXSUB+2));
589         MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
590                 thispe+NPEX, 0, comm, &request[1]);
591     }
592 }
593

```

```

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

```

```

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

```

```

701 long int ixbegin, ixend, jybegin, jyend;
702 realtype xfact, yfact, *udata, *iddata, dx, dy;
703
704 /* Initialize uu. */
705
706 udata = NV_DATA_P(uu);
707 iddata = NV_DATA_P(id);
708
709 /* Set mesh spacings and subgrid indices for this PE. */
710 dx = data->dx;
711 dy = data->dy;
712 ixsub = data->ixsub;
713 jysub = data->jysub;
714
715 /* Set beginning and ending locations in the global array corresponding
716    to the portion of that array assigned to this processor. */
717 ixbegin = MXSUB*ixsub;
718 ixend   = MXSUB*(ixsub+1) - 1;
719 jybegin = MYSUB*jysub;
720 jyend   = MYSUB*(jysub+1) - 1;
721
722 /* Loop over the local array, computing the initial profile value.
723    The global indices are (i,j) and the local indices are (iloc,jloc).
724    Also set the id vector to zero for boundary points, one otherwise. */
725
726 N_VConst(ONE,id);
727 for (j = jybegin, jloc = 0; j <= jyend; j++, jloc++) {
728     yfact = data->dy*j;
729     offset= jloc*MXSUB;
730     for (i = ixbegin, iloc = 0; i <= ixend; i++, iloc++) {
731         xfact = data->dx * i;
732         loc = offset + iloc;
733         udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
734         if (i == 0 || i == MX-1 || j == 0 || j == MY-1) iddata[loc] = ZERO;
735     }
736 }
737
738 /* Initialize up. */
739
740 N_VConst(ZERO, up); /* Initially set up = 0. */
741
742 /* resHeat sets res to negative of ODE RHS values at interior points. */
743 resHeat(ZERO, uu, up, res, data);
744
745 /* Copy -res into up to get correct initial up values. */
746 N_VScale(-ONE, res, up);
747
748 return(0);
749 }
750
751 /*
752 * Print first lines of output and table heading
753 */
754

```

```

755 static void PrintHeader(long int Neq, realtype rtol, realtype atol)
756 {
757     printf("\niheatpk: Heat equation, parallel example problem for IDA \n");
758     printf("        Discretized heat equation on 2D unit square. \n");
759     printf("        Zero boundary conditions,");
760     printf(" polynomial initial conditions.\n");
761     printf("        Mesh dimensions: %d x %d", MX, MY);
762     printf("        Total system size: %ld\n\n", Neq);
763     printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
764     printf("        Processor array: %d x %d\n", NPEX, NPEY);
765 #if defined(SUNDIALS_EXTENDED_PRECISION)
766     printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
767 #elif defined(SUNDIALS_DOUBLE_PRECISION)
768     printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
769 #else
770     printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
771 #endif
772     printf("Constraints set to force all solution components >= 0. \n");
773     printf("SUPPRESSALG = TRUE to suppress local error testing on ");
774     printf("all boundary components. \n");
775     printf("Linear solver: IDASPGMR  ");
776     printf("Preconditioner: diagonal elements only.\n");
777
778     /* Print output table heading and initial line of table. */
779     printf("\n  Output Summary (umax = max-norm of solution) \n\n");
780     printf("  time      umax      k  nst  nni  nli  nre  nreS      h      npe  nps\n");
781     printf("-----\n");
782 }
783
784 /*
785  * PrintOutput: print max norm of solution and current solver statistics
786  */
787
788 static void PrintOutput(int id, void *mem, realtype t, N_Vector uu)
789 {
790     realtype hused, umax;
791     long int nst, nni, nje, nre, nreS, nli, npe, nps;
792     int kused, ier;
793
794     umax = N_VMaxNorm(uu);
795
796     if (id == 0) {
797
798         ier = IDAGetLastOrder(mem, &kused);
799         check_flag(&ier, "IDAGetLastOrder", 1, id);
800         ier = IDAGetNumSteps(mem, &nst);
801         check_flag(&ier, "IDAGetNumSteps", 1, id);
802         ier = IDAGetNumNonlinSolvIters(mem, &nni);
803         check_flag(&ier, "IDAGetNumNonlinSolvIters", 1, id);
804         ier = IDAGetNumResEvals(mem, &nre);
805         check_flag(&ier, "IDAGetNumResEvals", 1, id);
806         ier = IDAGetLastStep(mem, &hused);
807         check_flag(&ier, "IDAGetLastStep", 1, id);
808         ier = IDASpgmrGetNumJtimesEvals(mem, &nje);

```

```

809     check_flag(&ier, "IDASpgmrGetNumJtimesEvals", 1, id);
810     ier = IDASpgmrGetNumLinIters(mem, &nli);
811     check_flag(&ier, "IDASpgmrGetNumLinIters", 1, id);
812     ier = IDASpgmrGetNumResEvals(mem, &nreS);
813     check_flag(&ier, "IDASpgmrGetNumResEvals", 1, id);
814     ier = IDASpgmrGetNumPrecEvals(mem, &npe);
815     check_flag(&ier, "IDASpgmrGetPrecEvals", 1, id);
816     ier = IDASpgmrGetNumPrecSolves(mem, &nps);
817     check_flag(&ier, "IDASpgmrGetNumPrecSolves", 1, id);
818
819     #if defined(SUNDIALS_EXTENDED_PRECISION)
820     printf(" %5.2Lf %13.5Le %d %3ld %3ld %3ld %4ld %4ld %9.2Le %3ld %3ld\n",
821           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
822     #elif defined(SUNDIALS_DOUBLE_PRECISION)
823     printf(" %5.2f %13.5le %d %3ld %3ld %3ld %4ld %4ld %9.2le %3ld %3ld\n",
824           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
825     #else
826     printf(" %5.2f %13.5e %d %3ld %3ld %3ld %4ld %4ld %9.2e %3ld %3ld\n",
827           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
828     #endif
829
830 }
831 }
832
833 /*
834  * Print some final integrator statistics
835  */
836
837 static void PrintFinalStats(void *mem)
838 {
839     long int netf, ncf, ncfl;
840
841     IDAGetNumErrTestFails(mem, &netf);
842     IDAGetNumNonlinSolvConvFails(mem, &ncf);
843     IDASpgmrGetNumConvFails(mem, &ncfl);
844
845     printf("\nError test failures           = %ld\n", netf);
846     printf("Nonlinear convergence failures = %ld\n", ncf);
847     printf("Linear convergence failures      = %ld\n", ncfl);
848 }
849
850 /*
851  * Check function return value...
852  *   opt == 0 means SUNDIALS function allocates memory so check if
853  *   returned NULL pointer
854  *   opt == 1 means SUNDIALS function returns a flag so check if
855  *   flag >= 0
856  *   opt == 2 means function allocates memory so check if returned
857  *   NULL pointer
858  */
859
860 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
861 {
862     int *errflag;

```

```

863
864 if (opt == 0 && flagvalue == NULL) {
865     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
866     fprintf(stderr,
867             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
868             id, funcname);
869     return(1);
870 } else if (opt == 1) {
871     /* Check if flag < 0 */
872     errflag = (int *) flagvalue;
873     if (*errflag < 0) {
874         fprintf(stderr,
875                 "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
876                 id, funcname, *errflag);
877         return(1);
878     }
879 } else if (opt == 2 && flagvalue == NULL) {
880     /* Check if function returned NULL pointer - no memory allocated */
881     fprintf(stderr,
882             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
883             id, funcname);
884     return(1);
885 }
886
887 return(0);
888 }

```

E Listing of iwebbbd.c

```

1  /*
2  * -----
3  * $Revision: 1.21.2.2 $
4  * $Date: 2005/04/04 22:36:42 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                 Radu Serban @ LLNL
8  * -----
9  * Example program for IDA: Food web, parallel, GMRES, IDABBD
10 * preconditioner.
11 *
12 * This example program for IDA uses IDASPGMR as the linear solver.
13 * It is written for a parallel computer system and uses the
14 * IDABBDPRE band-block-diagonal preconditioner module for the
15 * IDASPGMR package. It was originally run on a Sun SPARC cluster
16 * and used MPICH.
17 *
18 * The mathematical problem solved in this example is a DAE system
19 * that arises from a system of partial differential equations after
20 * spatial discretization. The PDE system is a food web population
21 * model, with predator-prey interaction and diffusion on the unit
22 * square in two dimensions. The dependent variable vector is:
23 *
24 *      1  2      ns
25 *   c = ( c , c , ..., c ) , ns = 2 * np
26 *
27 * and the PDE's are as follows:
28 *
29 *      i      i      i
30 *   dc /dt = d(i)*(c  + c ) + R (x,y,c)   (i = 1,...,np)
31 *                xx    yy      i
32 *
33 *      i      i
34 *   0 = d(i)*(c  + c ) + R (x,y,c)   (i = np+1,...,ns)
35 *                xx    yy      i
36 *
37 * where the reaction terms R are:
38 *
39 *      i      ns      j
40 *   R (x,y,c) = c  * (b(i) + sum a(i,j)*c )
41 *      i      j=1
42 *
43 * The number of species is ns = 2 * np, with the first np being
44 * prey and the last np being predators. The coefficients a(i,j),
45 * b(i), d(i) are:
46 *
47 *   a(i,i) = -AA (all i)
48 *   a(i,j) = -GG (i <= np , j > np)
49 *   a(i,j) = EE (i > np, j <= np)
50 *   all other a(i,j) = 0
51 *   b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i <= np)
52 *   b(i) = -BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i > np)

```

```

53 *   d(i) = DPREY  (i <= np)
54 *   d(i) = DPRED  (i > np)
55 *
56 * Note: The above equations are written in 1-based indices,
57 * whereas the code has 0-based indices, being written in C.
58 *
59 * The various scalar parameters required are set using '#define'
60 * statements or directly in routine InitUserData. In this program,
61 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
62 * normal derivative = 0.
63 *
64 * A polynomial in x and y is used to set the initial values of the
65 * first np variables (the prey variables) at each x,y location,
66 * while initial values for the remaining (predator) variables are
67 * set to a flat value, which is corrected by IDACalcIC.
68 *
69 * The PDEs are discretized by central differencing on a MX by MY
70 * mesh, and so the system size Neq is the product
71 * MX * MY * NUM_SPECIES. The system is actually implemented on
72 * submeshes, processor by processor, with an MXSUB by MYSUB mesh
73 * on each of NPEX * NPEY processors.
74 *
75 * The DAE system is solved by IDA using the IDASPGMR linear solver,
76 * in conjunction with the preconditioner module IDABBDPRE. The
77 * preconditioner uses a 5-diagonal band-block-diagonal
78 * approximation (half-bandwidths = 2). Output is printed at
79 * t = 0, .001, .01, .1, .4, .7, 1.
80 * -----
81 * References:
82 * [1] Peter N. Brown and Alan C. Hindmarsh,
83 *     Reduced Storage Matrix Methods in Stiff ODE systems,
84 *     Journal of Applied Mathematics and Computation, Vol. 31
85 *     (May 1989), pp. 40-91.
86 *
87 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
88 *     Using Krylov Methods in the Solution of Large-Scale
89 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
90 *     (1994), pp. 1467-1488.
91 *
92 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
93 *     Consistent Initial Condition Calculation for Differential-
94 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
95 *     pp. 1495-1512.
96 * -----
97 */
98
99 #include <stdio.h>
100 #include <stdlib.h>
101 #include <math.h>
102 #include "sundialstypes.h" /* Definitions of realtype and booleantype */
103 #include "iterative.h" /* Contains the types of preconditioning */
104 #include "ida.h" /* Main header file */
105 #include "idaspgmr.h" /* Use IDASPGMR linear solver */
106 #include "nvector_parallel.h" /* Definitions of type N_Vector, macro NV_DATA_P */

```

```

107 #include "sundialsmath.h"      /* Contains R.Sqrt routine          */
108 #include "smalldense.h"       /* Contains definitions for denalloc routine */
109 #include "mpi.h"              /* MPI library routines              */
110 #include "idabbdpre.h"        /* Definitions for the IDABBDPRE preconditioner */
111
112 /* Problem Constants. */
113
114 #define NPREY      1          /* Number of prey (= number of predators). */
115 #define NUM_SPECIES 2*NPREY
116
117 #define PI          RCONST(3.1415926535898) /* pi */
118 #define FOURPI     (RCONST(4.0)*PI)      /* 4 pi */
119
120 #define MXSUB      10        /* Number of x mesh points per processor subgrid */
121 #define MYSUB      10        /* Number of y mesh points per processor subgrid */
122 #define NPEX       2         /* Number of subgrids in the x direction */
123 #define NPEY       2         /* Number of subgrids in the y direction */
124 #define MX         (MXSUB*NPEX) /* MX = number of x mesh points */
125 #define MY         (MYSUB*NPEY) /* MY = number of y mesh points */
126 #define NSMXSUB    (NUM_SPECIES * MXSUB)
127 #define NEQ        (NUM_SPECIES*MX*MY) /* Number of equations in system */
128 #define AA         RCONST(1.0) /* Coefficient in above eqns. for a */
129 #define EE         RCONST(10000.) /* Coefficient in above eqns. for a */
130 #define GG         RCONST(0.5e-6) /* Coefficient in above eqns. for a */
131 #define BB         RCONST(1.0) /* Coefficient in above eqns. for b */
132 #define DPREY     RCONST(1.0) /* Coefficient in above eqns. for d */
133 #define DPRED     RCONST(0.05) /* Coefficient in above eqns. for d */
134 #define ALPHA     RCONST(50.) /* Coefficient alpha in above eqns. */
135 #define BETA      RCONST(1000.) /* Coefficient beta in above eqns. */
136 #define AX        RCONST(1.0) /* Total range of x variable */
137 #define AY        RCONST(1.0) /* Total range of y variable */
138 #define RTOL      RCONST(1.e-5) /* rtol tolerance */
139 #define ATOL      RCONST(1.e-5) /* atol tolerance */
140 #define ZERO      RCONST(0.) /* 0. */
141 #define ONE       RCONST(1.0) /* 1. */
142 #define NOUT      6
143 #define TMULT     RCONST(10.0) /* Multiplier for tout values */
144 #define TADD      RCONST(0.3) /* Increment for tout values */
145
146 /* User-defined vector accessor macro IJ_Vptr. */
147
148 /*
149  * IJ_Vptr is defined in order to express the underlying 3-d structure of the
150  * dependent variable vector from its underlying 1-d storage (an N_Vector).
151  * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
152  * species index is = 0, x-index ix = i, and y-index jy = j.
153  */
154
155 #define IJ_Vptr(vv,i,j) (&NV_Ith_P(vv, (i)*NUM_SPECIES + (j)*NSMXSUB ))
156
157 /* Type: UserData. Contains problem constants, preconditioner data, etc. */
158
159 typedef struct {
160     long int ns, np, thispe, npes, ixsub, jysub, npex, npey;

```

```

161     long int mxsub, mysub, nsmxsub, nsmxsub2;
162     realtype dx, dy, **acoef;
163     realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef [NUM_SPECIES],
164         rhs [NUM_SPECIES], cext [(MXSUB+2)*(MYSUB+2)*NUM_SPECIES];
165     MPI_Comm comm;
166     N_Vector rates;
167     long int n_local;
168 } *UserData;
169
170 /* Prototypes for functions called by the IDA Solver. */
171
172 static int resweb(realtype tt,
173                 N_Vector cc, N_Vector cp, N_Vector rr,
174                 void *res_data);
175
176 static int reslocal(long int Nlocal, realtype tt,
177                   N_Vector cc, N_Vector cp, N_Vector res,
178                   void *res_data);
179
180 static int rescomm(long int Nlocal, realtype tt,
181                  N_Vector cc, N_Vector cp,
182                  void *res_data);
183
184 /* Prototypes for supporting functions */
185
186 static void BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jsub,
187                 long int dsizex, long int dsizey, realtype carray[]);
188
189 static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
190                     long int ixsub, long int jsub,
191                     long int dsizex, long int dsizey,
192                     realtype cext[], realtype buffer[]);
193
194 static void BRecvWait(MPI_Request request[], long int ixsub, long int jsub,
195                     long int dsizex, realtype cext[], realtype buffer[]);
196
197 static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
198                   UserData webdata);
199
200 static realtype dotprod(long int size, realtype *x1, realtype *x2);
201
202 /* Prototypes for private functions */
203
204 static void InitUserData(UserData webdata, int thispe, int npes,
205                       MPI_Comm comm);
206
207 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
208                               N_Vector scrch, UserData webdata);
209
210 static void PrintHeader(long int SystemSize, int maxl,
211                       long int mudq, long int mldq,
212                       long int mukeep, long int mlkeep,
213                       realtype rtol, realtype atol);
214

```

```

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, t0, cc, cp, 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->mysub;
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     }

```

```

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 /*

```

```

539 * PrintOutput: Print output values at output time t = tt.
540 * Selected run statistics are printed. Then values of c1 and c2
541 * are printed for the bottom left and top right grid points only.
542 */
543
544 static void PrintOutput(void *mem, N_Vector cc, realtype tt,
545                        UserData webdata, MPI_Comm comm)
546 {
547     MPI_Status status;
548     realtype *cdata, c1ast[2], hused;
549     long int nst;
550     int i, kused, flag, thispe, npelast, ilast;;
551
552     thispe = webdata->thispe;
553     npelast = webdata->npes - 1;
554     cdata = NV_DATA_P(cc);
555
556     /* Send conc. at top right mesh point from PE npes-1 to PE 0. */
557     if (thispe == npelast) {
558         ilast = NUM_SPECIES*MXSUB*MYSUB - 2;
559         if (npelast != 0)
560             MPI_Send(&cdata[ilast], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
561         else { c1ast[0] = cdata[ilast]; c1ast[1] = cdata[ilast+1]; }
562     }
563
564     /* On PE 0, receive conc. at top right from PE npes - 1.
565        Then print performance data and sampled solution values. */
566
567     if (thispe == 0) {
568         if (npelast != 0)
569             MPI_Recv(&c1ast[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
570
571         flag = IDAGetLastOrder(mem, &kused);
572         check_flag(&flag, "IDAGetLastOrder", 1, thispe);
573         flag = IDAGetNumSteps(mem, &nst);
574         check_flag(&flag, "IDAGetNumSteps", 1, thispe);
575         flag = IDAGetLastStep(mem, &hused);
576         check_flag(&flag, "IDAGetLastStep", 1, thispe);
577
578     #if defined(SUNDIALS_EXTENDED_PRECISION)
579         printf("%8.2Le %12.4Le %12.4Le | %3ld %1d %12.4Le\n",
580              tt, cdata[0], c1ast[0], nst, kused, hused);
581         for (i=1; i<NUM_SPECIES; i++)
582             printf("          %12.4Le %12.4Le | \n", cdata[i], c1ast[i]);
583     #elif defined(SUNDIALS_DOUBLE_PRECISION)
584         printf("%8.2le %12.4le %12.4le | %3ld %1d %12.4le\n",
585              tt, cdata[0], c1ast[0], nst, kused, hused);
586         for (i=1; i<NUM_SPECIES; i++)
587             printf("          %12.4le %12.4le | \n", cdata[i], c1ast[i]);
588     #else
589         printf("%8.2e %12.4e %12.4e | %3ld %1d %12.4e\n",
590              tt, cdata[0], c1ast[0], nst, kused, hused);
591         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, ncf, 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, &ncf);
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", ncf);
643
644     printf("Number of linear iterations      = %ld\n", nli);
645     printf("Number of linear conv. failures  = %ld\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   nmysub = (webdata->ns)*(webdata->mysub);
761
762   /* Start receiving boundary data from neighboring PEs. */
763
764   BRecvPost(comm, request, thispe, ixsub, jysub, nsmxsub, nmysub,
765             cext, buffer);
766
767   /* Send data from boundary of local grid to neighboring PEs. */
768
769   BSend(comm, thispe, ixsub, jysub, nsmxsub, nmysub, 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 reatype 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 jsub,
830                     long int dsizey, reatype cext[], reatype buffer[])
831 {
832     int i;
833     long int ly, dsizey2, offsetce, offsetbuf;
834     reatype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
835     MPI_Status status;
836
837     dsizey2 = dsizey + 2*NUM_SPECIES;
838
839     /* If jsub > 0, receive data for bottom x-line of cext. */
840     if (jsub != 0)
841         MPI_Wait(&request[0], &status);
842
843     /* If jsub < NPEY-1, receive data for top x-line of cext. */
844     if (jsub != 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)*dsizey2;
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 jsub,
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 jsub > 0, send data from bottom x-line of cc. */
888
889     if (jsub != 0)
890         MPI_Send(&cdata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);
891
892     /* If jsub < NPEY-1, send data from top x-line of cc. */
893
894     if (jsub != 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[locce+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[locce+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[locce+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[locce+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
```