

Example Programs for IDA v2.4.0

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

November 6, 2006



UCRL-SM-208113

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: idadenx	3
2.2	A banded example: idabanx2	5
2.3	A Krylov example: idakryx	8
3	Parallel example problems	11
3.1	A user preconditioner example: idakryx1_p	11
3.2	An IDABBDPRE preconditioner example: idakryx2_bbd_p	12
4	Fortran example problems	15
4.1	A serial example: fidadenx	15
4.2	A parallel example: fidakryx_bbd_p	16
	References	19
A	Listing of idadenx.c	20
B	Listing of idabanx2.c	27
C	Listing of idakryx.c	39
D	Listing of idakryx1_p.c	49
E	Listing of idakryx2_bbd_p.c	65
F	Listing of fidadenx.f	84
G	Listing of fidakryx_bbd_p.f	89

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

- `idakryx1_p` solves the same 2-D heat equation problem as `idakryx`, with `IDASPGMR` in parallel, and with a user-supplied diagonal preconditioner,
- `idakryx1_bbd_p` solves the same problem as `idakryx1_p`.

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

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

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

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

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

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

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

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

In the descriptions below, we make frequent references to the IDA User Document [2]. All citations to specific sections (e.g. §5.2) are references to parts of that User Document, unless explicitly stated otherwise.

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

2 Serial example problems

2.1 A dense example: idadenx

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

$$\begin{cases} y_1' &= -.04y_1 + 10^4y_2y_3 \\ y_2' &= +.04y_1 - 10^4y_2y_3 - 3 \cdot 10^7y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1. \end{cases} \quad (1)$$

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 $t = 4 \cdot 10^{10}$. While integrating the system, the program also use the rootfinding feature to find the points at which $y_1 = 10^{-4}$ or at which $y_3 = 0.01$.

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

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

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

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

After various declarations, the `main` program begins by allocating memory for the `yy`, `yp`, and `avtol` vectors using `N_VNew_Serial` with a length argument of `NEQ` ($= 3$). The lines following that load the initial values of the 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 `avtol`. See §5.5.1 for full details of this call.

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

The calls to `IDADense` (see §5.5.3) and `IDADenseSetJacFn` (see §5.5.6.4) specify the IDADENSE linear solver with an analytic Jacobian supplied by the user-supplied function `jacrob`.

The actual solution of the DAE initial value problem is accomplished in the loop over values of the output time `tout`. In each pass of the loop, the program calls `IDASolve` in the `IDA_NORMAL` mode, meaning that the integrator is to take steps until it overshoots `tout` and then interpolate to $t = \text{tout}$, putting the computed value of $y(\text{tout})$ and $y'(\text{tout})$ into `yy` and `yp`, respectively, with `tret = tout`. The return value in this case is `IDA_SUCCESS`. However, if `IDASolve` finds a root before reaching the next value of `tout`, it returns `IDA_ROOT_RETURN` and stores the root location in `tret` and the solution there in `yy` and `yp`. In either case, the program prints `t` ($= \text{tret}$) and `yy`, and also the cumulative number of steps taken so far, and the current method order and step size. In the case of a root, the program calls `IDAGetRootInfo` to get a length-2 array `rootsfound` of bits showing which root function was found to have a root. If `IDASolve` returned any negative value (indicating a failure), the program breaks out of the loop. In the case of a `IDA_SUCCESS` return, the value of `tout` is advanced (multiplied by 10) and a counter (`iout`) is advanced, so that the loop can be ended when that counter reaches the preset number of output times, `NOUT = 12`. See §5.5.5 for full details of the call to `IDASolve`.

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

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

The functions `resrob` (of type `IDAResFn`) and `jacrob` (of type `IDADenseJacFn`) are straightforward expressions of the DAE system (1) and its system Jacobian. The function `jacrob` makes use of the macro `IJth` discussed above. See §5.6.1 for detailed specifications of `IDAResFn`. Similarly, the function `grob` defines the two functions, g_0 and g_1 , whose roots are to be found. See §5.7.2 for a detailed description of the `grob` function.

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

idadenx sample output						
<p>idadenx: Robertson kinetics DAE serial example problem for IDA Three equation chemical kinetics problem.</p> <p>Linear solver: IDADENSE, with user-supplied Jacobian. Tolerance parameters: rtol = 0.0001 atol = 1e-08 1e-14 1e-06 Initial conditions y0 = (1 0 0) Constraints and id not used.</p>						
t	y1	y2	y3		nst	k h
2.6403e-01	9.8997e-01	3.4706e-05	1.0000e-02		85	2 6.4537e-02
rootsfound[] = 0 1						
4.0000e-01	9.8517e-01	3.3864e-05	1.4796e-02		88	2 6.4537e-02
4.0000e+00	9.0550e-01	2.2403e-05	9.4473e-02		102	4 4.1426e-01
4.0000e+01	7.1582e-01	9.1851e-06	2.8417e-01		136	2 1.3422e+00
4.0000e+02	4.5049e-01	3.2226e-06	5.4950e-01		190	4 3.3557e+01
4.0000e+03	1.8321e-01	8.9429e-07	8.1679e-01		239	4 3.4533e+02
4.0000e+04	3.8984e-02	1.6218e-07	9.6102e-01		287	5 2.0140e+03
4.0000e+05	4.9389e-03	1.9852e-08	9.9506e-01		339	3 1.6788e+04
4.0000e+06	5.1683e-04	2.0684e-09	9.9948e-01		444	4 2.1755e+05
2.0793e+07	1.0000e-04	4.0004e-10	9.9990e-01		495	4 1.0146e+06
rootsfound[] = 1 0						
4.0000e+07	5.2036e-05	2.0816e-10	9.9995e-01		506	5 2.5503e+06
4.0000e+08	5.2103e-06	2.0841e-11	9.9999e-01		541	4 2.3847e+07
4.0000e+09	5.2125e-07	2.0850e-12	1.0000e-00		569	4 3.9351e+08
4.0000e+10	5.1091e-08	2.0437e-13	1.0000e-00		589	2 6.0246e+09
Final Run Statistics:						
Number of steps = 589						
Number of residual evaluations = 832						
Number of Jacobian evaluations = 79						
Number of nonlinear iterations = 832						
Number of error test failures = 14						
Number of nonlinear conv. failures = 0						
Number of root fn. evaluations = 617						

2.2 A banded example: idabanx2

This example is a model of a multi-species food web [1], in which predator-prey relationships with diffusion in a 2-D spatial domain are simulated. Here we consider a model with $s = 2p$ species: p predators and p prey. Species $1, \dots, p$ (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 `idabanx2.c` program (listed in Appendix B) includes the file `ida_band.h` in order to use the IDABAND linear solver. This file contains the prototype for the `IDABand` routine, the definition for the band matrix type `BandMat`, and the `BAND_COL` and `BAND_COL_ELEM` macros for accessing matrix elements. See §9.2. The main IDA header file `ida.h` is included for the prototypes of the solver user-callable functions and IDA constants, while the file `nvector_serial.h` is included for the definition of the serial `N_Vector` type. The header file `sundials_smalldense.h` is included for the `denalloc` function used in allocating memory for the user data structure.

The include lines at the top of the file are followed by definitions of problem constants which include the x and y mesh dimensions, `MX` and `MY`, the number of equations `NEQ`, the scalar relative and absolute tolerances `RTOL` and `ATOL`, and various parameters for the food-web problem.

Spatial discretization of the PDE naturally produces a DAE system in which equations are numbered by mesh coordinates (i, j) . The user-defined macro `IJth_Vptr` isolates the translation for the mathematical two-dimensional index to the one-dimensional `N_Vector` index and allows the user to write clean, readable code to access components of the dependent variable. `IJ_Vptr(v,i,j)` returns a pointer to the location in `v` corresponding to the species with index `is = 0`, x-index `ix = i`, and y-index `jy = j`.

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

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

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

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

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

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

Sample output from `idabanx2` follows.

```

----- idabanx2 sample output -----

idabanx2: Predator-prey DAE serial example problem for IDA

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

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

```

0.00e+00	1.0000e+01	1.0000e+05		0 0	1.6310e-08
	1.0000e+05	1.0000e+05			
1.00e-03	1.0318e+01	1.0822e+05		32 4	1.0823e-04
	1.0319e+05	1.0822e+05			
1.00e-02	1.6188e+02	1.9734e+06		127 4	1.4203e-04
	1.6189e+06	1.9734e+06			
1.00e-01	2.4019e+02	2.7072e+06		235 1	3.9159e-02
	2.4019e+06	2.7072e+06			
4.00e-01	2.4019e+02	2.7072e+06		238 1	3.1327e-01
	2.4019e+06	2.7072e+06			
7.00e-01	2.4019e+02	2.7072e+06		239 1	6.2654e-01
	2.4019e+06	2.7072e+06			
1.00e+00	2.4019e+02	2.7072e+06		239 1	6.2654e-01
	2.4019e+06	2.7072e+06			

Final run statistics:					
Number of steps				= 239	
Number of residual evaluations				= 3339	
Number of Jacobian evaluations				= 36	
Number of nonlinear iterations				= 421	
Number of error test failures				= 3	
Number of nonlinear conv. failures				= 0	

2.3 A Krylov example: idakryx

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

The domain is the unit square $\Omega = \{0 \leq x, y \leq 1\}$ 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, `ida_spgmr.h` is included for the definitions of constants and function prototypes associated with the SPGMR method.

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

The user-supplied preconditioner setup and solve functions, `PsetupHeat` and `PsolveHeat`, and the pointer to user data (`data`) are specified through a call to `IDASpilsSetPreconditioner`. In a loop over the desired output times, `IDASolve` is called in `IDA_NORMAL` mode and the maximum solution norm is printed.

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

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

The user-supplied functions `PsetupHeat` and `PsolveHeat` together define the left 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 `idakryx` follows.

idakryx sample output										
<pre> idakryx: Heat equation, serial example problem for IDA Discretized heat equation on 2D unit square. Zero boundary conditions, polynomial initial conditions. Mesh dimensions: 10 x 10 Total system size: 100 Tolerance parameters: rtol = 0 atol = 0.001 Constraints set to force all solution components >= 0. Linear solver: IDASPGMR, preconditioner using diagonal elements. Case 1: gsytpe = MODIFIED_GS Output Summary (umax = max-norm of solution) time umax k nst nni nje nre nreLS h npe nps ----- 0.01 8.24106e-01 2 12 14 7 14 7 2.56e-03 8 21 0.02 6.88134e-01 3 15 18 12 18 12 5.12e-03 8 30 0.04 4.70711e-01 3 18 24 21 24 21 6.58e-03 9 45 0.08 2.16509e-01 3 22 29 30 29 30 1.32e-02 9 59 0.16 4.57687e-02 4 28 36 44 36 44 1.32e-02 9 80 0.32 2.09938e-03 4 35 44 67 44 67 2.63e-02 10 111 0.64 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 </pre>										

Case 2: gstype = CLASSICAL_GS

Output Summary (umax = max-norm of solution)

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

Error test failures = 1

Nonlinear convergence failures = 0

Linear convergence failures = 0

3 Parallel example problems

3.1 A user preconditioner example: `idakryx1_p`

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

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

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

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

The `idakryx1_p` main program begins with MPI calls to initialize MPI and to set 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 `idakryx`, constraints are passed to IDA through the `N_Vector` `constraints` and the function `IDASetConstraints`. A temporary `N_Vector` `res` is also created here, for use only in `SetInitialProfiles`. All components of `constraints` are set to 1.0 indicating that non-negativity constraints are to be imposed on each solution component. In addition, for illustration purposes, `idakryx` also excludes the algebraic components of the solution (specified through the `N_Vector` `id`) from the error test by calling `IDASetSuppressAlg` with a flag `TRUE`.

Sample output from `idakryx1_p` follows.

```
----- idakryx1_p sample output -----  
  
idakryx1_p: Heat equation, parallel example problem for IDA  
            Discretized heat equation on 2D unit square.  
            Zero boundary conditions, polynomial initial conditions.  
            Mesh dimensions: 10 x 10           Total system size: 100  
  
Subgrid dimensions: 5 x 5           Processor array: 2 x 2  
Tolerance parameters:  rtol = 0      atol = 0.001  
Constraints set to force all solution components >= 0.
```

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

Output Summary (umax = max-norm of solution)

time	umax	k	nst	nni	nli	nre	nreLS	h	npe	nps
0.00	9.75461e-01	0	0	0	0	0	0	0.00e+00	0	0
0.01	8.24106e-01	2	12	14	7	14	7	2.56e-03	8	21
0.02	6.88134e-01	3	15	18	12	18	12	5.12e-03	8	30
0.04	4.70711e-01	3	18	24	21	24	21	6.58e-03	9	45
0.08	2.16509e-01	3	22	29	30	29	30	1.32e-02	9	59
0.16	4.57687e-02	4	28	36	44	36	44	1.32e-02	9	80
0.32	2.09938e-03	4	35	44	67	44	67	2.63e-02	10	111
0.64	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	1.50906e-19	1	44	56	77	56	77	3.37e+00	17	133
10.24	4.63224e-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: idakryx2_bbd_p

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

As with `idakryx1_p`, we use a $NPEX \times NPEY$ 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 `idakryx2_bbd_p` main program, following MPI initializations and creation of user data block `webdata` and `N_Vector` variables, the initial profiles are set, the IDA memory block is created and allocated, the IDABBDPRE preconditioner is initialized, and the IDASPGMR linear solver is attached to the IDA solver. The call to `IDACalcIC` corrects the initial values so that they are consistent with the DAE algebraic constraints.

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

Sample output from `idakryx2_bbd_p` follows.

```

----- idakryx2_bbd_p sample output -----

idakryx2_bbd_p: Predator-prey DAE parallel example problem

Number of species ns: 2
Mesh dimensions:      20 x 20
Total system size:    800
Subgrid dimensions:   10 x 10
Processor array:      2 x 2
Tolerance parameters:
  relative tolerance = 1e-05
  absolute tolerance = 1e-05
Linear solver: scaled preconditioned GMRES (IDASPGMR)
  max. Krylov dimension: maxl = 16
Preconditioner: band-block-diagonal (IDABBDPRE)
  mudq = 20, mldq = 20, mukeep = 2, mlkeep = 2
CalcIC called to correct initial predator concentrations

-----
  t          bottom-left  top-right  | nst  k      h
-----
0.00e+00    1.0000e+01    1.0000e+01  |   0   0    1.6310e-08
              1.0000e+05    1.0000e+05  |
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  | 123   3    1.9481e-04
              1.6189e+06    1.9735e+06  |
1.00e-01    2.4019e+02    2.7072e+02  | 197   1    4.0396e-02
              2.4019e+06    2.7072e+06  |
4.00e-01    2.4019e+02    2.7072e+02  | 200   1    3.2316e-01
              2.4019e+06    2.7072e+06  |
7.00e-01    2.4019e+02    2.7072e+02  | 200   1    3.2316e-01
              2.4019e+06    2.7072e+06  |
1.00e+00    2.4019e+02    2.7072e+02  | 201   1    6.4633e-01
              2.4019e+06    2.7072e+06  |
-----

Final statistics:

Number of steps                = 201
Number of residual evaluations = 1110
Number of nonlinear iterations = 245
Number of error test failures  = 0
Number of nonlinear conv. failures = 0

```

Number of linear iterations	= 863
Number of linear conv. failures	= 0
Number of preconditioner setups	= 26
Number of preconditioner solves	= 1110
Number of local residual evals.	= 1092

4 Fortran example problems

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

4.1 A serial example: fidadenx

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

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

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

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

```

fidadenx sample output

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

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

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

```

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

Final Run Statistics:

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

4.2 A parallel example: fidakryx_bbd_p

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

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

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

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

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

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

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

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

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

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

Output Summary

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

time	umax	k	nst	nni	nli	nre	nge	h	npe	nps
0.1000E-01	0.82411E+00	2	12	14	7	14+ 7	96	0.26E-02	8	21
0.2000E-01	0.68812E+00	3	15	18	12	18+12	96	0.51E-02	8	30
0.4000E-01	0.47075E+00	3	18	24	22	24+22	108	0.66E-02	9	46
0.8000E-01	0.21660E+00	3	22	29	30	29+30	108	0.13E-01	9	59
0.1600E+00	0.45659E-01	4	28	37	43	37+43	120	0.26E-01	10	80
0.3200E+00	0.21096E-02	4	35	45	59	45+59	120	0.24E-01	10	104
0.6400E+00	0.55368E-04	1	40	54	71	54+71	156	0.19E+00	13	125
0.1280E+01	0.15597E-18	1	42	56	71	56+71	180	0.76E+00	15	127
0.2560E+01	0.33865E-20	1	43	57	71	57+71	192	0.15E+01	16	128
0.5120E+01	0.86074E-20	1	44	58	71	58+71	204	0.30E+01	17	129
0.1024E+02	0.16630E-19	1	45	59	71	59+71	216	0.61E+01	18	130

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

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

Output Summary

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

time	umax	k	nst	nni	nli	nre	nge	h	npe	nps
0.1000E-01	0.82411E+00	2	12	14	7	14+ 7	32	0.26E-02	8	21
0.2000E-01	0.68812E+00	3	15	18	12	18+12	32	0.51E-02	8	30
0.4000E-01	0.47093E+00	3	19	23	20	23+20	36	0.10E-01	9	43
0.8000E-01	0.21655E+00	3	23	27	32	27+32	36	0.10E-01	9	59
0.1600E+00	0.45225E-01	4	27	33	44	33+44	40	0.20E-01	10	77
0.3200E+00	0.21868E-02	3	34	41	67	41+67	44	0.41E-01	11	108
0.6400E+00	0.48847E-18	1	39	49	86	49+86	52	0.16E+00	13	135

0.1280E+01	0.53982E-18	1	41	51	86	51+86	60	0.66E+00	15	137
0.2560E+01	0.74194E-17	1	42	52	86	52+86	64	0.13E+01	16	138
0.5120E+01	0.61081E-16	1	43	53	86	53+86	68	0.26E+01	17	139
0.1024E+02	0.40536E-15	1	44	54	86	54+86	72	0.52E+01	18	140
Error test failures = 0										
Nonlinear convergence failures = 0										
Linear convergence failures = 0										

References

- [1] Peter N. Brown. Decay to uniform states in food webs. *SIAM J. Appl. Math.*, 46:376–392, 1986.
- [2] A. C. Hindmarsh and R. Serban. User Documentation for IDA v2.4.0. Technical Report UCRL-SM-208112, LLNL, 2005.
- [3] H. H. Robertson. The solution of a set of reaction rate equations. In J. Walsh, editor, *Numerical analysis: an introduction*, pages 178–182. Academ. Press, 1966.

A Listing of idadenx.c

```
1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:09 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * This simple example problem for IDA, 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 * While integrating the system, we also use the rootfinding
21 * feature to find the points at which y1 = 1e-4 or at which
22 * y3 = 0.01.
23 *
24 * The problem is solved with IDA using IDADENSE for the linear
25 * solver, with a user-supplied Jacobian. Output is printed at
26 * t = .4, 4, 40, ..., 4e10.
27 * -----
28 */
29
30 #include <stdio.h>
31 #include <math.h>
32
33 #include <ida/ida.h>
34 #include <ida/ida_dense.h>
35 #include <nvector/nvector_serial.h>
36 #include <sundials/sundials_types.h>
37 #include <sundials/sundials_math.h>
38
39 /* Problem Constants */
40
41 #define NEQ      3
42 #define NOUT     12
43
44 #define ZERO  RCONST(0.0);
45 #define ONE   RCONST(1.0);
46
47 /* Macro to define dense matrix elements, indexed from 1. */
48
49 #define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1)
50
51 /* Prototypes of functions called by IDA */
52
53 int resrob(realtype tres, N_Vector yy, N_Vector yp,
54           N_Vector resval, void *rdata);
55
56 static int grob(realtype t, N_Vector yy, N_Vector yp,
57                realtype *gout, void *g_data);
```



```

58
59 int jacob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
60           N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
61           N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);
62
63 /* Prototypes of private functions */
64 static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y);
65 static void PrintOutput(void *mem, realtype t, N_Vector y);
66 static void PrintRootInfo(int root_f1, int root_f2);
67 static void PrintFinalStats(void *mem);
68 static int check_flag(void *flagvalue, char *funcname, int opt);
69
70 /*
71  *-----
72  * Main Program
73  *-----
74  */
75
76 int main(void)
77 {
78     void *mem;
79     N_Vector yy, yp, avtol;
80     realtype rtol, *yval, *ypval, *atval;
81     realtype t0, tout1, tout, tret;
82     int iout, retval, retvalr;
83     int rootsfound[2];
84
85     mem = NULL;
86     yy = yp = avtol = NULL;
87     yval = ypval = atval = NULL;
88
89     /* Allocate N-vectors. */
90     yy = N_VNew_Serial(NEQ);
91     if(check_flag((void *)yy, "N_VNew_Serial", 0)) return(1);
92     yp = N_VNew_Serial(NEQ);
93     if(check_flag((void *)yp, "N_VNew_Serial", 0)) return(1);
94     avtol = N_VNew_Serial(NEQ);
95     if(check_flag((void *)avtol, "N_VNew_Serial", 0)) return(1);
96
97     /* Create and initialize y, y', and absolute tolerance vectors. */
98     yval = NV_DATA_S(yy);
99     yval[0] = ONE;
100    yval[1] = ZERO;
101    yval[2] = ZERO;
102
103    ypval = NV_DATA_S(yp);
104    ypval[0] = RCONST(-0.04);
105    ypval[1] = RCONST(0.04);
106    ypval[2] = ZERO;
107
108    rtol = RCONST(1.0e-4);
109
110    atval = NV_DATA_S(avtol);
111    atval[0] = RCONST(1.0e-8);
112    atval[1] = RCONST(1.0e-14);
113    atval[2] = RCONST(1.0e-6);
114
115    /* Integration limits */
116    t0 = ZERO;

```

```

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

```

```

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

```



```

294 {
295     realtype *yval;
296     int retval, kused;
297     long int nst;
298     realtype hused;
299
300     yval = NV_DATA_S(y);
301
302     retval = IDAGetLastOrder(mem, &kused);
303     check_flag(&retval, "IDAGetLastOrder", 1);
304     retval = IDAGetNumSteps(mem, &nst);
305     check_flag(&retval, "IDAGetNumSteps", 1);
306     retval = IDAGetLastStep(mem, &hused);
307     check_flag(&retval, "IDAGetLastStep", 1);
308     #if defined(SUNDIALS_EXTENDED_PRECISION)
309     printf("%10.4Le_%12.4Le_%12.4Le_%12.4Le_%3ld_%1d_%12.4Le\n",
310           t, yval[0], yval[1], yval[2], nst, kused, hused);
311     #elif defined(SUNDIALS_DOUBLE_PRECISION)
312     printf("%10.4le_%12.4le_%12.4le_%12.4le_%3ld_%1d_%12.4le\n",
313           t, yval[0], yval[1], yval[2], nst, kused, hused);
314     #else
315     printf("%10.4e_%12.4e_%12.4e_%12.4e_%3ld_%1d_%12.4e\n",
316           t, yval[0], yval[1], yval[2], nst, kused, hused);
317     #endif
318 }
319
320 static void PrintRootInfo(int root_f1, int root_f2)
321 {
322     printf("rootsfound[]=%3d_%3d\n", root_f1, root_f2);
323     return;
324 }
325
326 /*
327  * Print final integrator statistics
328  */
329
330 static void PrintFinalStats(void *mem)
331 {
332     int retval;
333     long int nst, nni, nje, nre, nreLS, netf, ncnf, nge;
334
335     retval = IDAGetNumSteps(mem, &nst);
336     check_flag(&retval, "IDAGetNumSteps", 1);
337     retval = IDAGetNumResEvals(mem, &nre);
338     check_flag(&retval, "IDAGetNumResEvals", 1);
339     retval = IDADenseGetNumJacEvals(mem, &nje);
340     check_flag(&retval, "IDADenseGetNumJacEvals", 1);
341     retval = IDAGetNumNonlinSolvIters(mem, &nni);
342     check_flag(&retval, "IDAGetNumNonlinSolvIters", 1);
343     retval = IDAGetNumErrTestFails(mem, &netf);
344     check_flag(&retval, "IDAGetNumErrTestFails", 1);
345     retval = IDAGetNumNonlinSolvConvFails(mem, &ncnf);
346     check_flag(&retval, "IDAGetNumNonlinSolvConvFails", 1);
347     retval = IDADenseGetNumResEvals(mem, &nreLS);
348     check_flag(&retval, "IDADenseGetNumResEvals", 1);
349     retval = IDAGetNumGEvals(mem, &nge);
350     check_flag(&retval, "IDAGetNumGEvals", 1);
351
352     printf("\nFinalRunStatistics:\n\n");

```

```

353     printf("Number_of_steps=====_%ld\n", nst);
354     printf("Number_of_residual_evaluations=====_%ld\n", nre+nreLS);
355     printf("Number_of_Jacobian_evaluations=====_%ld\n", nje);
356     printf("Number_of_nonlinear_iterations=====_%ld\n", nni);
357     printf("Number_of_error_test_failures=====_%ld\n", netf);
358     printf("Number_of_nonlinear_conv._failures=_%ld\n", ncf);
359     printf("Number_of_root_fn._evaluations=====_%ld\n", nge);
360 }
361
362 /*
363  * Check function return value...
364  *   opt == 0 means SUNDIALS function allocates memory so check if
365  *       returned NULL pointer
366  *   opt == 1 means SUNDIALS function returns a flag so check if
367  *       flag >= 0
368  *   opt == 2 means function allocates memory so check if returned
369  *       NULL pointer
370  */
371
372 static int check_flag(void *flagvalue, char *funcname, int opt)
373 {
374     int *errflag;
375     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
376     if (opt == 0 && flagvalue == NULL) {
377         fprintf(stderr,
378             "\nSUNDIALS_ERROR:_%s()_failed_-_returned_NULL_pointer\n\n",
379             funcname);
380         return(1);
381     } else if (opt == 1) {
382         /* Check if flag < 0 */
383         errflag = (int *) flagvalue;
384         if (*errflag < 0) {
385             fprintf(stderr,
386                 "\nSUNDIALS_ERROR:_%s()_failed_with_flag=_%d\n\n",
387                 funcname, *errflag);
388             return(1);
389         }
390     } else if (opt == 2 && flagvalue == NULL) {
391         /* Check if function returned NULL pointer - no memory allocated */
392         fprintf(stderr,
393             "\nMEMORY_ERROR:_%s()_failed_-_returned_NULL_pointer\n\n",
394             funcname);
395         return(1);
396     }
397
398     return(0);
399 }

```

B Listing of idabanx2.c

```

1  /*
2  * -----
3  * $Revision: 1.3 $
4  * $Date: 2006/10/11 16:34:02 $
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
90 #include <ida/ida.h>           /* Main header file */
91 #include <ida/ida_band.h>      /* Use IDABAND linear solver */
92 #include <nvector/nvector_serial.h> /* Definitions of N_Vector and NV_DATA_S */
93 #include <sundials/sundials_smalldense.h> /* Definition of denalloc */
94 #include <sundials/sundials_types.h> /* Definitions of reatype and booleantype */
95
96 /* Problem Constants. */
97
98 #define NPREY          1          /* No. of prey (= no. of predators). */
99 #define NUM_SPECIES    2*NPREY
100
101 #define PI              RCONST(3.1415926535898)
102 #define FOURPI          (RCONST(4.0)*PI)
103
104 #define MX              20        /* MX = number of x mesh points */
105 #define MY              20        /* MY = number of y mesh points */
106 #define NSMX            (NUM_SPECIES * MX)
107 #define NEQ              (NUM_SPECIES*MX*MY)
108 #define AA              RCONST(1.0) /* Coefficient in above eqns. for a */
109 #define EE              RCONST(10000.) /* Coefficient in above eqns. for a */
110 #define GG              RCONST(0.5e-6) /* Coefficient in above eqns. for a */
111 #define BB              RCONST(1.0) /* Coefficient in above eqns. for b */
112 #define DPREY           RCONST(1.0) /* Coefficient in above eqns. for d */
113 #define DPRED           RCONST(0.05) /* Coefficient in above eqns. for d */
114 #define ALPHA           RCONST(50.) /* Coefficient alpha in above eqns. */
115 #define BETA            RCONST(1000.) /* Coefficient beta in above eqns. */
116 #define AX              RCONST(1.0) /* Total range of x variable */

```



```

117 #define AY          RCONST(1.0)      /* Total range of y variable      */
118 #define RTOL        RCONST(1.e-5)    /* Relative tolerance             */
119 #define ATOL        RCONST(1.e-5)    /* Absolute tolerance             */
120 #define NOUT        6                 /* Number of output times        */
121 #define TMULT        RCONST(10.0)     /* Multiplier for tout values     */
122 #define TADD         RCONST(0.3)     /* Increment for tout values     */
123 #define ZERO         RCONST(0.)
124 #define ONE          RCONST(1.0)
125
126 /*
127  * User-defined vector and accessor macro: IJ_Vptr.
128  * IJ_Vptr is defined in order to express the underlying 3-D structure of
129  * the dependent variable vector from its underlying 1-D storage (an N_Vector).
130  * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
131  * species index is = 0, x-index ix = i, and y-index jy = j.
132  */
133
134 #define IJ_Vptr(vv,i,j) (&NV_Ith_S(vv, (i)*NUM_SPECIES + (j)*NSMX))
135
136 /* Type: UserData.  Contains problem constants, etc. */
137
138 typedef struct {
139     long int Neq, ns, np, mx, my;
140     realtype dx, dy, **acoeff;
141     realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES];
142     N_Vector rates;
143 } *UserData;
144
145 /* Prototypes for functions called by the IDA Solver. */
146
147 static int resweb(realtype time, N_Vector cc, N_Vector cp, N_Vector resval,
148                 void *rdata);
149
150 /* Prototypes for private Helper Functions. */
151
152 static void InitUserData(UserData webdata);
153 static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
154                               UserData webdata);
155 static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol);
156 static void PrintOutput(void *mem, N_Vector c, realtype t);
157 static void PrintFinalStats(void *mem);
158 static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate, UserData webdata);
159 static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
160                     UserData webdata);
161 static realtype dotprod(long int size, realtype *x1, realtype *x2);
162 static int check_flag(void *flagvalue, char *funcname, int opt);
163
164 /*
165  *-----
166  * MAIN PROGRAM
167  *-----
168  */
169
170 int main()
171 {
172     void *mem;
173     UserData webdata;
174     N_Vector cc, cp, id;
175     int iout, retval;

```

```

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

```

```

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

```

```

294 static int resweb(realtype tt, N_Vector cc, N_Vector cp,
295                  N_Vector res, void *rdata)
296 {
297     long int jx, jy, is, yloc, loc, np;
298     realtype *resv, *cpv;
299     UserData webdata;
300
301     webdata = (UserData)rdata;
302
303     cpv = NV_DATA_S(cp);
304     resv = NV_DATA_S(res);
305     np = webdata->np;
306
307     /* Call Fweb to set res to vector of right-hand sides. */
308     Fweb(tt, cc, res, webdata);
309
310     /* Loop over all grid points, setting residual values appropriately
311        for differential or algebraic components. */
312
313     for (jy = 0; jy < MY; jy++) {
314         yloc = NSMX * jy;
315         for (jx = 0; jx < MX; jx++) {
316             loc = yloc + NUM_SPECIES * jx;
317             for (is = 0; is < NUM_SPECIES; is++) {
318                 if (is < np)
319                     resv[loc+is] = cpv[loc+is] - resv[loc+is];
320                 else
321                     resv[loc+is] = -resv[loc+is];
322             }
323         }
324     }
325
326     return(0);
327 }
328
329 /*
330 -----
331 * 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 = NPRES;
348     webdata->dx = AX/(MX-1);
349     webdata->dy = AY/(MY-1);
350     webdata->Neq= NEQ;
351
352

```

```

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

```

```

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

```

```

471  */
472
473 static void PrintOutput(void *mem, N_Vector c, realtype t)
474 {
475     int i, kused, flag;
476     long int nst;
477     realtype *c_bl, *c_tr, hused;
478
479     flag = IDAGetLastOrder(mem, &kused);
480     check_flag(&flag, "IDAGetLastOrder", 1);
481     flag = IDAGetNumSteps(mem, &nst);
482     check_flag(&flag, "IDAGetNumSteps", 1);
483     flag = IDAGetLastStep(mem, &hused);
484     check_flag(&flag, "IDAGetLastStep", 1);
485
486     c_bl = IJ_Vptr(c,0,0);
487     c_tr = IJ_Vptr(c,MX-1,MY-1);
488
489     #if defined(SUNDIALS_EXTENDED_PRECISION)
490     printf("%8.2Le%12.4Le%12.4Le%3ld%1d%12.4Le\n",
491           t, c_bl[0], c_tr[1], nst, kused, hused);
492     for (i=1;i<NUM_SPECIES;i++)
493         printf("%%%%%%%%%12.4Le%12.4Le%3ld%1d%12.4Le\n",c_bl[i],c_tr[i]);
494     #elif defined(SUNDIALS_DOUBLE_PRECISION)
495     printf("%8.2le%12.4le%12.4le%3ld%1d%12.4le\n",
496           t, c_bl[0], c_tr[1], nst, kused, hused);
497     for (i=1;i<NUM_SPECIES;i++)
498         printf("%%%%%%%%%12.4le%12.4le%3ld%1d%12.4le\n",c_bl[i],c_tr[i]);
499     #else
500     printf("%8.2e%12.4e%12.4e%3ld%1d%12.4e\n",
501           t, c_bl[0], c_tr[1], nst, kused, hused);
502     for (i=1;i<NUM_SPECIES;i++)
503         printf("%%%%%%%%%12.4e%12.4e%3ld%1d%12.4e\n",c_bl[i],c_tr[i]);
504     #endif
505
506     printf("\n");
507 }
508
509 /*
510  * PrintFinalStats: Print final run data contained in iopt.
511  */
512
513 static void PrintFinalStats(void *mem)
514 {
515     long int nst, nre, nreLS, nni, nje, netf, ncfn;
516     int flag;
517
518     flag = IDAGetNumSteps(mem, &nst);
519     check_flag(&flag, "IDAGetNumSteps", 1);
520     flag = IDAGetNumNonlinSolvIters(mem, &nni);
521     check_flag(&flag, "IDAGetNumNonlinSolvIters", 1);
522     flag = IDAGetNumResEvals(mem, &nre);
523     check_flag(&flag, "IDAGetNumResEvals", 1);
524     flag = IDAGetNumErrTestFails(mem, &netf);
525     check_flag(&flag, "IDAGetNumErrTestFails", 1);
526     flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
527     check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1);
528     flag = IDABandGetNumJacEvals(mem, &nje);
529     check_flag(&flag, "IDABandGetNumJacEvals", 1);

```

```

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

```



```

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

```

```

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

```

C Listing of idakryx.c

```

1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:09 $
5  * -----
6  * Programmer(s): Allan Taylor, Alan Hindmarsh and
7  *                Radu Serban @ LLNL
8  * -----
9  * Example problem for IDA: 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
38 #include <ida/ida.h>
39 #include <ida/ida_spgmr.h>
40 #include <nvector/nvector_serial.h>
41 #include <sundials/sundials_types.h>
42
43 /* Problem Constants */
44
45 #define NOUT 11
46 #define MGRID 10
47 #define NEQ MGRID*MGRID
48 #define ZERO RCONST(0.0)
49 #define ONE RCONST(1.0)
50 #define TWO RCONST(2.0)
51 #define FOUR RCONST(4.0)
52
53 /* User data type */
54
55 typedef struct {
56     long int mm; /* number of grid points */
57     realtype dx;

```

```

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

```

```

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

```

```

176     */
177
178     /* Print case number, output table heading, and initial line of table. */
179
180     printf("\n\nCase_1:gsytpe=MODIFIED_GS\n");
181     printf("\n\nOutput_Summary_(umax=umax-norm_of_solution)_\n\n");
182     printf("timeumaxkinstnnijnereLSnpe_nps\n" );
183     printf("-----\n");
184
185     /* Loop over output times, call IDASolve, and print results. */
186
187     for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
188         ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
189         if(check_flag(&ier, "IDASolve", 1)) return(1);
190         PrintOutput(mem, tret, uu);
191     }
192
193     /* Print remaining counters. */
194
195     ier = IDAGetNumErrTestFails(mem, &netf);
196     check_flag(&ier, "IDAGetNumErrTestFails", 1);
197
198     ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
199     check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);
200
201     ier = IDASpilsGetNumConvFails(mem, &ncfl);
202     check_flag(&ier, "IDASpilsGetNumConvFails", 1);
203
204     printf("\nError_test_failures=_%ld\n", netf);
205     printf("Nonlinear_convergence_failures=_%ld\n", ncfn);
206     printf("Linear_convergence_failures=_%ld\n", ncfl);
207
208     /*
209     * -----
210     * CASE II
211     * -----
212     */
213
214     /* Re-initialize uu, up. */
215
216     SetInitialProfile(data, uu, up, res);
217
218     /* Re-initialize IDA and IDASPGMR */
219
220     ier = IDAREinit(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
221     if(check_flag(&ier, "IDAREinit", 1)) return(1);
222
223     ier = IDASpilsSetGSType(mem, CLASSICAL_GS);
224     if(check_flag(&ier, "IDASpilsSetGSType",1)) return(1);
225
226     /* Print case number, output table heading, and initial line of table. */
227
228     printf("\n\nCase_2:gsytpe=CLASSICAL_GS\n");
229     printf("\n\nOutput_Summary_(umax=umax-norm_of_solution)_\n\n");
230     printf("timeumaxkinstnnijnereLSnpe_nps\n" );
231     printf("-----\n");
232
233     /* Loop over output times, call IDASolve, and print results. */
234

```

```

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

```

```

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

```



```

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

```

```

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

```

```

471     long int nst, nni, nje, nre, nreLS, nli, npe, nps;
472     int kused, ier;
473
474     umax = N_VMaxNorm(uu);
475
476     ier = IDAGetLastOrder(mem, &kused);
477     check_flag(&ier, "IDAGetLastOrder", 1);
478     ier = IDAGetNumSteps(mem, &nst);
479     check_flag(&ier, "IDAGetNumSteps", 1);
480     ier = IDAGetNumNonlinSolvIters(mem, &nni);
481     check_flag(&ier, "IDAGetNumNonlinSolvIters", 1);
482     ier = IDAGetNumResEvals(mem, &nre);
483     check_flag(&ier, "IDAGetNumResEvals", 1);
484     ier = IDAGetLastStep(mem, &hused);
485     check_flag(&ier, "IDAGetLastStep", 1);
486     ier = IDASpilsGetNumJtimesEvals(mem, &nje);
487     check_flag(&ier, "IDASpilsGetNumJtimesEvals", 1);
488     ier = IDASpilsGetNumLinIters(mem, &nli);
489     check_flag(&ier, "IDASpilsGetNumLinIters", 1);
490     ier = IDASpilsGetNumResEvals(mem, &nreLS);
491     check_flag(&ier, "IDASpilsGetNumResEvals", 1);
492     ier = IDASpilsGetNumPrecEvals(mem, &npe);
493     check_flag(&ier, "IDASpilsGetPrecEvals", 1);
494     ier = IDASpilsGetNumPrecSolves(mem, &nps);
495     check_flag(&ier, "IDASpilsGetNumPrecSolves", 1);
496
497     #if defined(SUNDIALS_EXTENDED_PRECISION)
498     printf("_%5.2Lf_%13.5Le_%d_%3ld_%3ld_%3ld_%4ld_%4ld_%9.2Le_%3ld_%3ld\n",
499           t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
500     #elif defined(SUNDIALS_DOUBLE_PRECISION)
501     printf("_%5.2f_%13.5le_%d_%3ld_%3ld_%3ld_%4ld_%4ld_%9.2le_%3ld_%3ld\n",
502           t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
503     #else
504     printf("_%5.2f_%13.5e_%d_%3ld_%3ld_%3ld_%4ld_%4ld_%9.2e_%3ld_%3ld\n",
505           t, umax, kused, nst, nni, nje, nre, nreLS, hused, npe, nps);
506     #endif
507 }
508
509 /*
510  * Check function return value...
511  *   opt == 0 means SUNDIALS function allocates memory so check if
512  *               returned NULL pointer
513  *   opt == 1 means SUNDIALS function returns a flag so check if
514  *               flag >= 0
515  *   opt == 2 means function allocates memory so check if returned
516  *               NULL pointer
517  */
518
519 static int check_flag(void *flagvalue, char *funcname, int opt)
520 {
521     int *errflag;
522
523     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
524     if (opt == 0 && flagvalue == NULL) {
525         fprintf(stderr,
526             "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
527             funcname);
528         return(1);
529     } else if (opt == 1) {

```

```

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

```

D Listing of idakryx1.p.c

```

1  /*
2  * -----
3  * $Revision: 1.1 $
4  * $Date: 2006/07/05 15:50:08 $
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
41 #include <ida/ida.h>          /* Main header file */
42 #include <ida/ida_spgmr.h>    /* Use IDASPGMR linear solver */
43 #include <nvector/nvector_parallel.h> /* Definitions of N_Vector and NV_DATA_P */
44 #include <sundials/sundials_types.h> /* Definitions of realtype and booleantype */
45 #include <sundials/sundials_math.h> /* Contains RSqrt routine */
46
47 #include <mpi.h>              /* MPI library routines */
48
49 #define ZERO    RCONST(0.0)
50 #define ONE     RCONST(1.0)
51 #define TWO     RCONST(2.0)
52
53 #define NOUT      11          /* Number of output times */
54
55 #define NPEX      2           /* No. PEs in x direction of PE array */
56 #define NPEY      2           /* No. PEs in y direction of PE array */
57                               /* Total no. PEs = NPEX*NPEY */

```

```

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

```

```

117 static void PrintOutput(int id, void *mem, realtype t, N_Vector uu);
118
119 static void PrintFinalStats(void *mem);
120
121 static int check_flag(void *flagvalue, char *funcname, int opt, int id);
122
123 /*
124  *-----
125  * MAIN PROGRAM
126  *-----
127  */
128
129 int main(int argc, char *argv[])
130 {
131     MPI_Comm comm;
132     void *mem;
133     UserData data;
134     int iout, thispe, ier, npes;
135     long int Neq, local_N;
136     realtype rtol, atol, t0, t1, tout, tret;
137     N_Vector uu, up, constraints, id, res;
138
139     mem = NULL;
140     data = NULL;
141     uu = up = constraints = id = res = NULL;
142
143     /* Get processor number and total number of pe's. */
144
145     MPI_Init(&argc, &argv);
146     comm = MPI_COMM_WORLD;
147     MPI_Comm_size(comm, &npes);
148     MPI_Comm_rank(comm, &thispe);
149
150     if (npes != NPEX*NPEY) {
151         if (thispe == 0)
152             fprintf(stderr,
153                 "\nMPI_ERROR(0): npes=%d is not equal to NPEX*NPEY=%d\n",
154                 npes, NPEX*NPEY);
155         MPI_Finalize();
156         return(1);
157     }
158
159     /* Set local length local_N and global length Neq. */
160
161     local_N = MXSUB*MYSUB;
162     Neq      = MX * MY;
163
164     /* Allocate and initialize the data structure and N-vectors. */
165
166     data = (UserData) malloc(sizeof *data);
167     data->pp = NULL;
168     if(check_flag((void *)data, "malloc", 2, thispe))
169         MPI_Abort(comm, 1);
170
171     uu = N_VNew_Parallel(comm, local_N, Neq);
172     if(check_flag((void *)uu, "N_VNew_Parallel", 0, thispe))
173         MPI_Abort(comm, 1);
174
175     up = N_VNew_Parallel(comm, local_N, Neq);

```

```

176     if(check_flag((void *)up, "N_VNew_Parallel", 0, thispe))
177         MPI_Abort(comm, 1);
178
179     res = N_VNew_Parallel(comm, local_N, Neq);
180     if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe))
181         MPI_Abort(comm, 1);
182
183     constraints = N_VNew_Parallel(comm, local_N, Neq);
184     if(check_flag((void *)constraints, "N_VNew_Parallel", 0, thispe))
185         MPI_Abort(comm, 1);
186
187     id = N_VNew_Parallel(comm, local_N, Neq);
188     if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe))
189         MPI_Abort(comm, 1);
190
191     /* An N-vector to hold preconditioner. */
192     data->pp = N_VNew_Parallel(comm, local_N, Neq);
193     if(check_flag((void *)data->pp, "N_VNew_Parallel", 0, thispe))
194         MPI_Abort(comm, 1);
195
196     InitUserData(thispe, comm, data);
197
198     /* Initialize the uu, up, id, and res profiles. */
199
200     SetInitialProfile(uu, up, id, res, data);
201
202     /* Set constraints to all 1's for nonnegative solution values. */
203
204     N_VConst(ONE, constraints);
205
206     t0 = ZERO; t1 = RCONST(0.01);
207
208     /* Scalar relative and absolute tolerance. */
209
210     rtol = ZERO;
211     atol = RCONST(1.0e-3);
212
213     /* Call IDACreate and IDAMalloc to initialize solution. */
214
215     mem = IDACreate();
216     if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);
217
218     ier = IDASetRdata(mem, data);
219     if(check_flag(&ier, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);
220
221     ier = IDASetSuppressAlg(mem, TRUE);
222     if(check_flag(&ier, "IDASetSuppressAlg", 1, thispe)) MPI_Abort(comm, 1);
223
224     ier = IDASetId(mem, id);
225     if(check_flag(&ier, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);
226
227     ier = IDASetConstraints(mem, constraints);
228     if(check_flag(&ier, "IDASetConstraints", 1, thispe)) MPI_Abort(comm, 1);
229     N_VDestroy_Parallel(constraints);
230
231     ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
232     if(check_flag(&ier, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
233
234     /* Call IDASpgmr to specify the linear solver. */

```



```

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

```

```

294  * This parallel implementation uses several supporting routines.
295  * First a call is made to rescomm to do communication of subgrid boundary
296  * data into array uext. Then reslocal is called to compute the residual
297  * on individual processors and their corresponding domains. The routines
298  * BSend, BRecvPost, and BRecvWait handle interprocessor communication
299  * of uu required to calculate the residual.
300  */
301
302  int resHeat(realtype tt,
303             N_Vector uu, N_Vector up, N_Vector rr,
304             void *res_data)
305  {
306      int retval;
307
308      /* Call rescomm to do inter-processor communication. */
309      retval = rescomm(uu, up, res_data);
310
311      /* Call reslocal to calculate res. */
312      retval = reslocal(tt, uu, up, rr, res_data);
313
314      return(0);
315  }
316
317  /*
318  * PsetupHeat: setup for diagonal preconditioner for heatsk.
319  *
320  * The optional user-supplied functions PsetupHeat and
321  * PsolveHeat together must define the left preconditioner
322  * 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;

```

```

353     npex  = data->npex;
354     npey  = data->npey;
355
356     /* Initially set all pp elements to one. */
357     N_VConst(ONE, data->pp);
358
359     /* Prepare to loop over subgrid. */
360     ixbegin = 0;
361     ixend   = mxsub-1;
362     jybegin = 0;
363     jyend   = mysub-1;
364     if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
365     if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
366     pelinv = ONE/(c_j + data->coeffxy);
367
368     /* Load the inverse of the preconditioner diagonal elements
369        in loop over all the local subgrid. */
370
371     for (ly = jybegin; ly <=jyend; ly++) {
372         for (lx = ixbegin; lx <= ixend; lx++) {
373             locu = lx + ly*mxsub;
374             ppv[locu] = pelinv;
375         }
376     }
377
378     return(0);
379 }
380
381 /*
382 * 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 * SUPPORTING FUNCTIONS
406 -----
407 */
408
409
410
411

```

```

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

```

```

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

```

```

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

```

```

589     offsetue = (1 + (MYSUB+1)*(MXSUB+2));
590     MPI_Irecv(&uext[offsetue], dsizey, PVEC_REAL_MPI_TYPE,
591             thispe+NPEX, 0, comm, &request[1]);
592 }
593
594 /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
595 if (ixsub != 0) {
596     MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
597             thispe-1, 0, comm, &request[2]);
598 }
599
600 /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
601 if (ixsub != NPEX-1) {
602     MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
603             thispe+1, 0, comm, &request[3]);
604 }
605
606 return(0);
607
608 }
609
610 /*
611  * Routine to finish receiving boundary data from neighboring PEs.
612  * Notes:
613  * 1) buffer should be able to hold 2*MYSUB realtype entries, should be
614  *    passed to both the BRecvPost and BRecvWait functions, and should not
615  *    be manipulated between the two calls.
616  * 2) request should have four entries, and should be passed in both
617  *    calls also.
618  */
619
620 static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
621                     long int dsizey, realtype uext[], realtype buffer[])
622 {
623     long int ly, dsizey2, offsetue;
624     realtype *bufleft = buffer, *bufright = buffer+MYSUB;
625     MPI_Status status;
626
627     dsizey2 = dsizey + 2;
628
629     /* If jysub > 0, receive data for bottom x-line of uext. */
630     if (jysub != 0)
631         MPI_Wait(&request[0], &status);
632
633     /* If jysub < NPEY-1, receive data for top x-line of uext. */
634     if (jysub != NPEY-1)
635         MPI_Wait(&request[1], &status);
636
637     /* If ixsub > 0, receive data for left y-line of uext (via bufleft). */
638     if (ixsub != 0) {
639         MPI_Wait(&request[2], &status);
640
641         /* Copy the buffer to uext. */
642         for (ly = 0; ly < MYSUB; ly++) {
643             offsetue = (ly+1)*dsizey2;
644             uext[offsetue] = bufleft[ly];
645         }
646     }
647 }

```

```

648     /* If ixsub < NPEX-1, receive data for right y-line of uext (via bufright). */
649     if (ixsub != NPEX-1) {
650         MPI_Wait(&request[3], &status);
651
652         /* Copy the buffer to uext */
653         for (ly = 0; ly < MYSUB; ly++) {
654             offsetue = (ly+2)*dsizex2 - 1;
655             uext[offsetue] = bufright[ly];
656         }
657     }
658
659     return(0);
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->mxcsub = MXSUB;
687     data->mysub = MYSUB;
688     data->comm = comm;
689     return(0);
690 }
691
692 /*
693  * SetInitialProfile sets the initial values for the problem.
694  */
695
696 static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
697                             N_Vector res, UserData data)
698 {
699     long int i, iloc, j, jloc, offset, loc, ixsub, jysub;
700     long int ixbegin, ixend, jybegin, jyend;
701     realtype xfact, yfact, *udata, *iddata, dx, dy;
702
703     /* Initialize uu. */
704
705
706

```



```

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

```

```

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

```

```

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

```

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

E Listing of idakryx2_bbd_p.c

```

1  /*
2  * -----
3  * $Revision: 1.5 $
4  * $Date: 2006/10/11 16:34:01 $
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) = DPRED (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
103 #include <ida/ida.h>           /* Main header file */
104 #include <ida/ida_spgmr.h>     /* Use IDASPGMR linear solver */
105 #include <ida/ida_bbdpre.h>    /* Definitions for the IDABBDPRE prec. */
106 #include <nvector/nvector_parallel.h> /* Definitions of N_Vector and NV_DATA_P */
107 #include <sundials/sundials_smalldense.h> /* definition of denalloc */
108 #include <sundials/sundials_types.h> /* Definitions of reatype and booleantype */
109 #include <sundials/sundials_math.h> /* Contains RSqrt routine */
110
111 #include <mpi.h>               /* MPI library routines */
112
113 /* Problem Constants */
114
115 #define NPREDY          1      /* Number of prey (= number of predators). */
116 #define NUM_SPECIES 2*NPREDY

```

```

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

```

```

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

```



```

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

```

```

294     rtol = RTOL;
295     atol = ATOL;
296
297     /* Call IDACreate and IDAMalloc to initialize solution */
298
299     mem = IDACreate();
300     if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);
301
302     retval = IDASetRdata(mem, webdata);
303     if(check_flag(&retval, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);
304
305     retval = IDASetId(mem, id);
306     if(check_flag(&retval, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);
307
308     retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, rtol, &atol);
309     if(check_flag(&retval, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);
310
311     /* Call IDABBDPrecAlloc to initialize the band-block-diagonal preconditioner.
312        The half-bandwidths for the difference quotient evaluation are exact
313        for the system Jacobian, but only a 5-diagonal band matrix is retained. */
314
315     mudq = mldq = NSMXSUB;
316     mukeep = mlkeep = 2;
317     P_data = IDABBDPrecAlloc(mem, local_N, mudq, mldq, mukeep, mlkeep,
318                             ZERO, reslocal, NULL);
319     if(check_flag((void *)P_data, "IDABBDPrecAlloc", 0, thispe)) MPI_Abort(comm, 1);
320
321     /* Call IDABBDSPgmr to specify the IDA linear solver IDASPGMR and specify
322        the preconditioner routines supplied
323        maxl (max. Krylov subspace dim.) is set to 16. */
324
325     maxl = 16;
326     retval = IDABBDSPgmr(mem, maxl, P_data);
327     if(check_flag(&retval, "IDABBDSPgmr", 1, thispe)) MPI_Abort(comm, 1);
328
329     /* Call IDACalcIC (with default options) to correct the initial values. */
330
331     tout = RCONST(0.001);
332     retval = IDACalcIC(mem, IDA_YA_YDP_INIT, tout);
333     if(check_flag(&retval, "IDACalcIC", 1, thispe)) MPI_Abort(comm, 1);
334
335     /* On PE 0, print heading, basic parameters, initial values. */
336
337     if (thispe == 0) PrintHeader(SystemSize, maxl,
338                                mudq, mldq, mukeep, mlkeep,
339                                rtol, atol);
340     PrintOutput(mem, cc, t0, webdata, comm);
341
342     /* Call IDA in tout loop, normal mode, and print selected output. */
343
344     for (iout = 1; iout <= NOUT; iout++) {
345
346         retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
347         if(check_flag(&retval, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);
348
349         PrintOutput(mem, cc, tret, webdata, comm);
350
351         if (iout < 3) tout *= TMULT;
352         else          tout += TADD;

```

```

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

```

```

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

```

```

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

```

```

530 #endif
531 printf("Linear solver: scaled preconditioned GMRES (IDASPGMR)\n");
532 printf("max. Krylov dimension: maxl=%d\n", maxl);
533 printf("Preconditioner: band-block-diagonal (IDABBDPRE)\n");
534 printf("mudq=%ld, mldq=%ld, mukeep=%ld, mlkeep=%ld\n",
535        mudq, mldq, mukeep, mlkeep);
536 printf("CalcIC called to correct initial predator concentrations\n\n");
537 printf("-----\n");
538 printf("tt bottom-left top-right");
539 printf("nst kused h\n");
540 printf("-----\n\n");
541 }
542
543
544 /*
545  * PrintOutput: Print output values at output time t = tt.
546  * Selected run statistics are printed. Then values of c1 and c2
547  * are printed for the bottom left and top right grid points only.
548  */
549
550 static void PrintOutput(void *mem, N_Vector cc, realtype tt,
551                        UserData webdata, MPI_Comm comm)
552 {
553     MPI_Status status;
554     realtype *cdata, clast[2], hused;
555     long int nst;
556     int i, kused, flag, thispe, npelast, ilast;;
557
558     thispe = webdata->thispe;
559     npelast = webdata->npes - 1;
560     cdata = NV_DATA_P(cc);
561
562     /* Send conc. at top right mesh point from PE npes-1 to PE 0. */
563     if (thispe == npelast) {
564         ilast = NUM_SPECIES*MXSUB*MYSUB - 2;
565         if (npelast != 0)
566             MPI_Send(&cdata[ilast], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
567         else { clast[0] = cdata[ilast]; clast[1] = cdata[ilast+1]; }
568     }
569
570     /* On PE 0, receive conc. at top right from PE npes - 1.
571        Then print performance data and sampled solution values. */
572
573     if (thispe == 0) {
574
575         if (npelast != 0)
576             MPI_Recv(&clast[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
577
578         flag = IDAGetLastOrder(mem, &kused);
579         check_flag(&flag, "IDAGetLastOrder", 1, thispe);
580         flag = IDAGetNumSteps(mem, &nst);
581         check_flag(&flag, "IDAGetNumSteps", 1, thispe);
582         flag = IDAGetLastStep(mem, &hused);
583         check_flag(&flag, "IDAGetLastStep", 1, thispe);
584
585         #if defined(SUNDIALS_EXTENDED_PRECISION)
586         printf("%8.2Le%12.4Le%12.4Le%3ld%ld%12.4Le\n",
587                tt, cdata[0], clast[0], nst, kused, hused);
588         for (i=1; i<NUM_SPECIES; i++)

```

```

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

```

```

648     printf("Number_of_nonlinear_conv_failures=%ld\n\n", ncfn);
649
650     printf("Number_of_linear_iterations=%ld\n", nli);
651     printf("Number_of_linear_conv_failures=%ld\n\n", ncfl);
652
653     printf("Number_of_preconditioner_setups=%ld\n", npe);
654     printf("Number_of_preconditioner_solves=%ld\n", nps);
655     printf("Number_of_local_residual_evals=%ld\n", nge);
656
657 }
658
659 /*
660  * Check function return value...
661  *   opt == 0 means SUNDIALS function allocates memory so check if
662  *   returned NULL pointer
663  *   opt == 1 means SUNDIALS function returns a flag so check if
664  *   flag >= 0
665  *   opt == 2 means function allocates memory so check if returned
666  *   NULL pointer
667  */
668
669 static int check_flag(void *flagvalue, char *funcname, int opt, int id)
670 {
671     int *errflag;
672
673     if (opt == 0 && flagvalue == NULL) {
674         /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
675         fprintf(stderr,
676             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
677             id, funcname);
678         return(1);
679     } else if (opt == 1) {
680         /* Check if flag < 0 */
681         errflag = (int *) flagvalue;
682         if (*errflag < 0) {
683             fprintf(stderr,
684                 "\nSUNDIALS_ERROR(%d): %s() failed with flag=%d\n\n",
685                 id, funcname, *errflag);
686             return(1);
687         }
688     } else if (opt == 2 && flagvalue == NULL) {
689         /* Check if function returned NULL pointer - no memory allocated */
690         fprintf(stderr,
691             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
692             id, funcname);
693         return(1);
694     }
695
696     return(0);
697 }
698
699 /*
700  *-----
701  * FUNCTIONS CALLED BY IDA & SUPPORTING FUNCTIONS
702  *-----
703  */
704
705 /*
706  * resweb: System residual function for predator-prey system.

```



```

707  * To compute the residual function F, this routine calls:
708  * rescomm, for needed communication, and then
709  * reslocal, for computation of the residuals on this processor.
710  */
711
712  static int resweb(realtype tt,
713                  N_Vector cc, N_Vector cp, N_Vector rr,
714                  void *res_data)
715  {
716      int retval;
717      UserData webdata;
718      long int Nlocal;
719
720      webdata = (UserData) res_data;
721
722      Nlocal = webdata->n_local;
723
724      /* Call rescomm to do inter-processor communication. */
725      retval = rescomm(Nlocal, tt, cc, cp, res_data);
726
727      /* Call reslocal to calculate the local portion of residual vector. */
728      retval = reslocal(Nlocal, tt, cc, cp, rr, res_data);
729
730      return(0);
731  }
732
733  /*
734   * rescomm: Communication routine in support of resweb.
735   * This routine performs all inter-processor communication of components
736   * of the cc vector needed to calculate F, namely the components at all
737   * interior subgrid boundaries (ghost cell data). It loads this data
738   * into a work array cext (the local portion of c, extended).
739   * The message-passing uses blocking sends, non-blocking receives,
740   * and receive-waiting, in routines BRecvPost, BSend, BRecvWait.
741   */
742
743  static int rescomm(long int Nlocal, realtype tt,
744                  N_Vector cc, N_Vector cp,
745                  void *res_data)
746  {
747
748      UserData webdata;
749      realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
750      long int thispe, ixsub, jysub, nsmxsub, nsmysub;
751      MPI_Comm comm;
752      MPI_Request request[4];
753
754      webdata = (UserData) res_data;
755      cdata = NV_DATA_P(cc);
756
757      /* Get comm, thispe, subgrid indices, data sizes, extended array cext. */
758
759      comm = webdata->comm;
760      thispe = webdata->thispe;
761
762      ixsub = webdata->ixsub;
763      jysub = webdata->jysub;
764      cext = webdata->cext;
765      nsmxsub = webdata->nsmxsub;

```

```

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

```

```

825 }
826
827 /*
828  * BRecvWait: Finish receiving boundary data from neighboring PEs.
829  * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype entries,
830  *     should be passed to both the BRecvPost and BRecvWait functions, and
831  *     should not be manipulated between the two calls.
832  * (2) request should have 4 entries, and is also passed in both calls.
833  */
834
835 static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
836                      long int dsize, realtype cext[], realtype buffer[])
837 {
838     int i;
839     long int ly, dsize2, offsetc, offsetb;
840     realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
841     MPI_Status status;
842
843     dsize2 = dsize + 2*NUM_SPECIES;
844
845     /* If jysub > 0, receive data for bottom x-line of cext. */
846     if (jysub != 0)
847         MPI_Wait(&request[0], &status);
848
849     /* If jysub < NPEY-1, receive data for top x-line of cext. */
850     if (jysub != NPEY-1)
851         MPI_Wait(&request[1], &status);
852
853     /* If ixsub > 0, receive data for left y-line of cext (via bufleft). */
854     if (ixsub != 0) {
855         MPI_Wait(&request[2], &status);
856
857         /* Copy the buffer to cext */
858         for (ly = 0; ly < MYSUB; ly++) {
859             offsetb = ly*NUM_SPECIES;
860             offsetc = (ly+1)*dsize2;
861             for (i = 0; i < NUM_SPECIES; i++)
862                 cext[offsetc+i] = bufleft[offsetb+i];
863         }
864     }
865
866     /* If ixsub < NPEX-1, receive data for right y-line of cext (via bufright). */
867     if (ixsub != NPEX-1) {
868         MPI_Wait(&request[3], &status);
869
870         /* Copy the buffer to cext */
871         for (ly = 0; ly < MYSUB; ly++) {
872             offsetb = ly*NUM_SPECIES;
873             offsetc = (ly+2)*dsize2 - NUM_SPECIES;
874             for (i = 0; i < NUM_SPECIES; i++)
875                 cext[offsetc+i] = bufright[offsetb+i];
876         }
877     }
878 }
879
880 /*
881  * BSend: Send boundary data to neighboring PEs.
882  * This routine sends components of cc from internal subgrid boundaries
883  * to the appropriate neighbor PEs.

```

```

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

```

```

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

```

```

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

```

```

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

```

F Listing of fidadenx.f

```

1  c -----
2  c $Revision: 1.1 $
3  c $Date: 2006/07/05 15:50:08 $
4  c -----
5  c This simple example problem for FIDA, due to Robertson, is from
6  c chemical kinetics, and consists of the following three equations:
7  c
8  c      dy1/dt = -.04*y1 + 1.e4*y2*y3
9  c      dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*y2**2
10 c      0      = y1 + y2 + y3 - 1
11 c
12 c on the interval from t = 0.0 to t = 4.e10, with initial
13 c conditions: y1 = 1, y2 = y3 = 0.
14 c
15 c While integrating the system, we also use the rootfinding feature
16 c to find the points at which y1 = 1.e-4 or at which y3 = 0.01.
17 c
18 c The problem is solved using a dense linear solver, with a
19 c user-supplied Jacobian. Output is printed at
20 c t = .4, 4, 40, ..., 4e10.
21 c -----
22 c
23 c program fidadenx
24 c
25 c implicit none
26 c
27 c integer ier, ierroot, info(2)
28 c integer*4 iout(25), ipar
29 c double precision rout(10), rpar
30 c
31 c integer iatol, nout, jout, itask
32 c integer nst, kused, hused
33 c integer*4 neq, i
34 c double precision t0, t1, rtol, tout, tret
35 c double precision y(3), yp(3), atol(3)
36 c
37 c data nst/3/, kused/9/, hused/2/
38 c
39 c Initialize variables
40 c
41 c      neq = 3
42 c      nout = 12
43 c      rtol = 1.0d-4
44 c      t0 = 0.0d0
45 c      t1 = 0.4d0
46 c      iatol = 2
47 c      itask = 1
48 c
49 c      y(1) = 1.0d0
50 c      y(2) = 0.0d0
51 c      y(3) = 0.0d0
52 c
53 c      yp(1) = -0.04d0
54 c      yp(2) = 0.04d0
55 c      yp(3) = 0.0d0
56 c
57 c      atol(1) = 1.0d-6

```



```

58         atol(2) = 1.0d-10
59         atol(3) = 1.0d-6
60     c
61     c Initialize IDA vector environment
62     c
63         call fnvinits(2, neq, ier)
64         if (ier .ne. 0) then
65             write(6,10) ier
66 10         format(///' SUNDIALS_ERROR: FNVINITS returned IER = ', i5)
67             stop
68         endif
69     c
70         call fidamalloc(t0, y, yp, iatol, rtol, atol,
71 &             iout, rout, ipar, rpar, ier)
72         if (ier .ne. 0) then
73             write(6,20) ier
74 20         format(///' SUNDIALS_ERROR: FIDAMALLOC returned IER = ', i5)
75             stop
76         endif
77     c
78     c Initialize rootfinding problem
79
80         call fidarootinit(2, ier)
81         if (ier .ne. 0) then
82             write(6,25) ier
83 25         format(///' SUNDIALS_ERROR: FIDAROOTINIT returned IER = ', i5)
84             call fidafree
85             stop
86         endif
87     c
88     c Attach dense linear solver
89     c
90         call fidadense(neq, ier)
91         call fidadensesetjac(1, ier)
92     c
93     c Print header
94     c
95         call prntintro(rtol, atol, y)
96     c
97         tout = t1
98     c
99     c
100        jout = 1
101        do while(jout .le. nout)
102    c
103            call fidasolve(tout, tret, y, yp, itask, ier)
104    c
105            write(6,40) tret, (y(i), i = 1,3), iout(nst), iout(kused),
106 &                rout(hused)
107 40        format(e10.4, 3(1x,e12.4), i5, i3, e12.4)
108    c
109            if (ier .lt. 0) then
110                write(6,50) ier, iout(15)
111 50        format(///' SUNDIALS_ERROR: FIDASOLVE returned IER = ',i5,/,
112 1          ' Linear Solver returned IER = ',i5)
113                call fidarootfree
114                call fidafree
115                stop
116            endif

```

```

117 c
118     if (ier .eq. 2) then
119         call fidarootinfo(2, info, ierroot)
120         if (ierroot .lt. 0) then
121             write(6,55) ier
122 55         format(///' SUNDIALS_ERROR: FIDAROOTINFO returned IER = ',
123 1             i5)
124             call fidarootfree
125             call fidadfree
126             stop
127         endif
128         write(6,60) (info(i), i = 1,2)
129 60         format(5x, 'Above is a root, INFO() = ', 2i3)
130     endif
131 c
132     if (ier .eq. 0) then
133         tout = tout * 10.0d0
134         jout = jout + 1
135     endif
136 c
137     ENDDO
138 c
139 c Print final statistics
140 c
141     call prntstats(iout)
142 c
143 c Free IDA memory
144 c
145     call fidarootfree
146     call fidadfree
147 c
148     stop
149     end
150 c
151 c =====
152 c
153     subroutine fidaresfun(tres, y, yp, res, ipar, rpar, reserr)
154 c
155     implicit none
156 c
157     integer reserr
158     integer*4 ipar(*)
159     double precision tres, rpar(*)
160     double precision y(*), yp(*), res(*)
161 c
162     res(1) = -0.04d0*y(1)+1.0d4*y(2)*y(3)
163     res(2) = -res(1)-3.0d7*y(2)*y(2)-yp(2)
164     res(1) = res(1)-yp(1)
165     res(3) = y(1)+y(2)+y(3)-1.0d0
166 c
167     reserr = 0
168 c
169     return
170     end
171 c
172 c =====
173 c
174     subroutine fidadjac(neq, t, y, yp, r, jac, cj, ewt, h,
175 1             ipar, rpar, wk1, wk2, wk3, djacerr)

```

```

176 C
177     implicit none
178 C
179     integer*4 neq, ipar(*)
180     integer djacerr
181     double precision t, h, cj, rpar(*)
182     double precision y(*), yp(*), r(*), ewt(*), jac(neq,neq)
183     double precision wk1(*), wk2(*), wk3(*)
184 C
185     jac(1,1) = -0.04d0-cj
186     jac(2,1) = 0.04d0
187     jac(3,1) = 1.0d0
188     jac(1,2) = 1.0d4*y(3)
189     jac(2,2) = -1.0d4*y(3)-6.0d7*y(2)-cj
190     jac(3,2) = 1.0d0
191     jac(1,3) = 1.0d4*y(2)
192     jac(2,3) = -1.0d4*y(2)
193     jac(3,3) = 1.0d0
194 C
195     djacerr = 0
196
197     return
198 end
199 C
200 C =====
201 C
202     subroutine fidarootfn(t, y, yp, g, ipar, rpar, ier)
203 C Fortran routine for rootfinding
204     implicit none
205 C
206     INTEGER*4 ipar(*), ier
207     double precision t, y(*), yp(*), g(*), rpar(*)
208 C
209     g(1) = y(1) - 1.0d-4
210     g(2) = y(3) - 1.0d-2
211
212     ier = 0
213
214     return
215 end
216 C
217 C =====
218 C
219     subroutine prntintro(rtol, atol, y)
220 C
221     implicit none
222 C
223     integer*4 i
224     double precision rtol, atol(*), y(*)
225 C
226     write(6,60) rtol, (atol(i), i = 1,3), (y(i), i = 1,3)
227 60 format(/'fidadenx: Robertson kinetics DAE serial example',
228 &        'problem for IDA', /, '          Three equation chemical',
229 &        'kinetics problem.', //,
230 &        'Tolerance parameters:  rtol = ', e8.2,
231 &        '      atol = ', 3(1x,e8.2), /,
232 &        'Initial conditions y0 = (', 3(1x,e8.2), ')', //,
233 &        '  t          y1          y2          y3          nst',
234 &        '  k          h')

```

```

235 C
236     return
237 end
238 C
239 C =====
240 C
241     subroutine prntstats(iout)
242 C
243     implicit none
244 C
245     integer*4 iout(25)
246     integer nst, reseval, jaceval, nni, ncf, netf, nge
247 C
248     data nst/3/, reseval/4/, jaceval/17/, nni/7/, netf/5/,
249 &        ncf/6/, nge/12/
250 C
251     write(6,70) iout(nst), iout(reseval), iout(jaceval),
252 &               iout(nni), iout(netf), iout(ncf), iout(nge)
253 70    format(/'Final Run Statistics:', //,
254 &           'Number of steps                = ', i3, /,
255 &           'Number of residual evaluations  = ', i3, /,
256 &           'Number of Jacobian evaluations  = ', i3, /,
257 &           'Number of nonlinear iterations  = ', i3, /,
258 &           'Number of error test failures   = ', i3, /,
259 &           'Number of nonlinear conv. failures = ', i3, /,
260 &           'Number of root function evals.   = ', i3)
261 C
262     return
263 end

```

G Listing of fidakryx_bbd_p.f

```

1  c -----
2  c $Revision: 1.2 $
3  c $Date: 2006/10/19 22:13:15 $
4  c -----
5  c Example problem for FIDA: 2D heat equation, parallel, GMRES,
6  c IDABBDPRE.
7  c
8  c This example solves a discretized 2D heat equation problem.
9  c This version uses the Krylov solver IDASPGMR and BBD
10 c preconditioning.
11 c
12 c The DAE system solved is a spatial discretization of the PDE
13 c  $du/dt = d^2u/dx^2 + d^2u/dy^2$ 
14 c on the unit square. The boundary condition is  $u = 0$  on all edges.
15 c Initial conditions are given by  $u = 16 x (1 - x) y (1 - y)$ . The
16 c PDE is treated with central differences on a uniform  $MX \times MY$ 
17 c grid. The values of  $u$  at the interior points satisfy ODEs, and
18 c equations  $u = 0$  at the boundaries are appended, to form a DAE
19 c system of size  $N = MX * MY$ . Here  $MX = MY = 10$ .
20 c
21 c The system is actually implemented on submeshes, processor by
22 c processor, with an  $MXSUB$  by  $MYSUB$  mesh on each of  $NPEX * NPEY$ 
23 c processors.
24 c
25 c The system is solved with FIDA using the Krylov linear solver
26 c IDASPGMR in conjunction with the preconditioner module IDABBDPRE.
27 c The preconditioner uses a tridiagonal approximation
28 c (half-bandwidths = 1). The constraints  $u \geq 0$  are posed for all
29 c components. Local error testing on the boundary values is
30 c suppressed. Output is taken at  $t = 0, .01, .02, .04, \dots, 10.24$ .
31 c -----
32 c
33 c program fidakryx_bbd_p
34 c
35 c include "mpif.h"
36 c
37 c global variables
38 c
39 c integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
40 c integer*4 ixsub, jysub
41 c integer thispe
42 c integer mxsubg, mysubg, nlocalg
43 c parameter (mxsubg = 5, mysubg = 5)
44 c parameter (nlocalg = mxsubg*mysubg)
45 c double precision dx, dy, coeffx, coeffy, coeffxy
46 c double precision uext((mxsubg+2)*(mysubg+2))
47 c
48 c local variables
49 c
50 c integer*4 mudq, mldq, mukeep, mlkeep
51 c integer*4 iout(25), ipar
52 c double precision rout(10), rpar
53 c integer nout, ier
54 c parameter (nout = 11)
55 c integer npes, inopt, maxl, gstype, maxrs, itask, iatol
56 c double precision t0, t1, tout, tret, dqrely, eplifac, dqincfac
57 c double precision atol, rtol

```

```

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

```

```

117 c
118     if (npes .ne. npex*npey) then
119         if (thispe .eq. 0) then
120             write(*,5) npes, npex*npey
121 5             format(///' MPI_ERROR: npes = ', i5, ' is not equal to ',
122 &                 'NPEX*NPEY = ', i5)
123             call mpi_finalize(ier)
124             stop
125         endif
126     endif
127 c
128     call fnvinitp(mpi_comm_world, 2, nlocal, neq, ier)
129     if (ier .ne. 0) then
130         write(*,6) ier
131 6         format(///' SUNDIALS_ERROR: FNVINITP returned IER = ', i5)
132         call mpi_finalize(ier)
133         stop
134     endif
135 c
136     jysub = int(thispe/npex)
137     ixsub = thispe-jysub*npex
138 c
139 c Initialize problem data
140 c
141     call setinitprofile(uu, up, id, res, constr, ipar, rpar)
142 c
143 c Initialize IDA environment
144 c
145     call fidamalloc(t0, uu, up, iatol, rtol, atol,
146 &                 iout, rout, ipar, rpar, ier)
147     if (ier .ne. 0) then
148         write(*,7) ier
149 7         format(///' SUNDIALS_ERROR: FIDAMALLOC returned IER = ', i5)
150         call mpi_abort(mpi_comm_world, 1, ier)
151         stop
152     endif
153 c
154 c Set optional inputs
155 c
156     call fidasetiin('SUPPRESS_ALG', 1, ier)
157     call fidasetvin('ID_VEC', id, ier)
158     call fidasetvin('CONSTR_VEC', constr, ier)
159 c
160 c Initialize and attach BBDSPGMR module
161 c
162 c
163     call fidabbddinit(nlocal, mudq, mldq, mukeep, mlkeep, dqrely, ier)
164     if (ier .ne. 0) then
165         write(*,8) ier
166 8         format(///' SUNDIALS_ERROR: FIDABBDINIT returned IER = ', i5)
167         call mpi_abort(mpi_comm_world, 1, ier)
168         stop
169     endif
170 c
171     call fidabbdspgmr(maxl, gstype, maxrs, eplifac, dqincfac, ier)
172     if (ier .ne. 0) then
173         write(*,9) ier
174 9         format(///' SUNDIALS_ERROR: FIDABBDSPGMR returned IER = ', i5)
175         call mpi_abort(mpi_comm_world, 1, ier)

```

```

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

```



```

235         if (thispe .eq. 0) then
236             call prntcase(2, mudq, mukeep)
237         endif
238     c
239     tout = t1
240     do 12 jout = 1, nout
241     c
242         call fidasolve(tout, tret, uu, up, itask, ier)
243     c
244         call prntoutput(tret, uu, iout, rout)
245     c
246         if (ier .ne. 0) then
247             write(*,13) ier
248     13         format(///' SUNDIALS_ERROR: FIDASOLVE returned IER = ', i5)
249             call fidafree
250             stop
251         endif
252     c
253         tout = tout*2.0d0
254     c
255     12 continue
256     c
257     c Print statistics
258     c
259         if (thispe .eq. 0) then
260             call prntfinalstats(iout)
261         endif
262     c
263     c Free memory
264     c
265         call fidabddfrees
266         call fidafrees
267     c
268         call mpi_finalize(ier)
269     c
270         stop
271     end
272     c
273     c =====
274     c
275         subroutine setinitprofile(uu, up, id, res, constr, ipar, rpar)
276     c
277     c global variables
278     c
279         integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
280         integer*4 ixsub, jysub, ipar(*)
281         integer thispe
282         integer mxsubg, mysubg, nlocalg
283         parameter (mxsubg = 5, mysubg = 5)
284         parameter (nlocalg = mxsubg*mysubg)
285         double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
286         double precision uext((mxsubg+2)*(mysubg+2))
287     c
288     c local variables
289     c
290         integer*4 i, iloc, j, jloc, offset, loc
291         integer*4 ixbegin, ixend, jybegin, jyend
292         integer reserr
293         double precision xfact, yfact

```

```

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

```

```

353 c
354 c global variables
355 c
356     integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
357     integer*4 ixsub, jysub, ipar(*)
358     integer thispe
359     integer mxsubg, mysubg, nlocalg
360     parameter (mxsubg = 5, mysubg = 5)
361     parameter (nlocalg = mxsubg*mysubg)
362     double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
363     double precision uext((mxsubg+2)*(mysubg+2))
364 c
365 c local variables
366 c
367     integer reserr
368     double precision tres
369     double precision u(*), up(*), res(*)
370 c
371     common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
372     &                nlocal, neq, mx, my, mxsub, mysub, npey, npex,
373     &                ixsub, jysub, thispe
374 c
375     call fidacommfn(nlocal, tres, u, up, ipar, rpar, reserr)
376 c
377     call fidaglocfn(nlocal, tres, u, up, res, ipar, rpar, reserr)
378 c
379     return
380 end
381 c
382 c =====
383 c
384     subroutine fidacommfn(nloc, tres, u, up, ipar, rpar, reserr)
385 c
386     include "mpif.h"
387 c
388 c global variables
389 c
390     integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
391     integer*4 ixsub, jysub, ipar(*)
392     integer thispe
393     integer mxsubg, mysubg, nlocalg
394     parameter (mxsubg = 5, mysubg = 5)
395     parameter (nlocalg = mxsubg*mysubg)
396     double precision dx, dy, coeffx, coeffy, coeffxy, rpar(*)
397     double precision uext((mxsubg+2)*(mysubg+2))
398 c
399 c local variables
400 c
401     integer*4 nloc
402     integer reserr
403     double precision tres, u(*), up(*)
404 c
405     integer request(mpi_status_size)
406     double precision buffer(2*mysub)
407 c
408     common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
409     &                nlocal, neq, mx, my, mxsub, mysub, npey, npex,
410     &                ixsub, jysub, thispe
411 c

```

```

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

```

```

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

```

```

530  C
531      if (jysub .ne. npey-1) then
532          offsetu = (mysub-1)*dsizex
533          call mpi_send(uarray(offsetu+1), dsizex, mpi_double_precision,
534      &                  thispe+npex, 0, mpi_comm_world, ier)
535      endif
536  C
537      if (ixsub .ne. 0) then
538          do 24 ly = 0, mysub-1
539              offsetu = ly*dsizex
540              bufleft(ly+1) = uarray(offsetu+1)
541      24      continue
542          call mpi_send(bufleft(1), dsizey, mpi_double_precision,
543      &                  thispe-1, 0, mpi_comm_world, ier)
544      endif
545  C
546      if (ixsub .ne. npex-1) then
547          do 25 ly = 0, mysub-1
548              offsetu = ly*mxsub+(mxsub-1)
549              bufright(ly+1) = uarray(offsetu+1)
550      25      continue
551          call mpi_send(bufright(1), dsizey, mpi_double_precision,
552      &                  thispe+1, 0, mpi_comm_world, ier)
553      endif
554  C
555      return
556  end
557  C
558  C =====
559  C
560      subroutine brecvpost(request, dsizex, dsizey, buffer)
561  C
562      include "mpif.h"
563  C
564  C global variables
565  C
566      integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
567      integer*4 ixsub, jysub
568      integer thispe
569      integer mxsubg, mysubg, nlocalg
570      parameter (mxsubg = 5, mysubg = 5)
571      parameter (nlocalg = mxsubg*mysubg)
572      double precision dx, dy, coeffx, coeffy, coeffxy
573      double precision uext((mxsubg+2)*(mysubg+2))
574  C
575  C local variables
576  C
577      integer*4 dsizex, dsizey
578      integer request(*)
579      double precision buffer(*)
580  C
581      integer ier
582      integer*4 offsetue
583  C
584      common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
585      &                  nlocal, neq, mx, my, mxsub, mysub, npey, npex,
586      &                  ixsub, jysub, thispe
587  C
588      if (jysub .ne. 0) then

```

```

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

```

```

648         if (jysub .ne. 0) then
649             call mpi_wait(request(1), status, ier)
650         endif
651     c
652         if (jysub .ne. npey-1) then
653             call mpi_wait(request(2), status, ier)
654         endif
655     c
656         if (ixsub .ne. 0) then
657             call mpi_wait(request(3), status, ier)
658             do 26 ly = 0, mysub-1
659                 offsetue = (ly+1)*dsizex2
660                 uext(offsetue+1) = buffer(ly+1)
661         26         continue
662         endif
663     c
664         if (ixsub .ne. npex-1) then
665             call mpi_wait(request(4), status, ier)
666             do 27 ly = 0, mysub-1
667                 offsetue = (ly+2)*dsizex2-1
668                 uext(offsetue+1) = buffer(ly+mysub+1)
669         27         continue
670         endif
671     c
672         return
673     end
674     c
675     c =====
676     c
677         subroutine prntoutput(tret, u, iout, rout)
678     c
679     c global variables
680     c
681         integer*4 nlocal, neq, npex, npey, mxsub, mysub, mx, my
682         integer*4 ixsub, jysub
683         integer thispe
684         integer mxsubg, mysubg, nlocalg
685         parameter (mxsubg = 5, mysubg = 5)
686         parameter (nlocalg = mxsubg*mysubg)
687         double precision dx, dy, coeffx, coeffy, coeffxy
688         double precision uext((mxsubg+2)*(mysubg+2))
689     c
690     c local variables
691     c
692         integer*4 iout(*), lenrwbbd, leniwbbd, ngebbd
693         double precision tret, umax, u(*), rout(*)
694     c
695         common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
696         & nlocal, neq, mx, my, mxsub, mysub, npey, npex,
697         & ixsub, jysub, thispe
698     c
699         call maxnorm(u, umax)
700     c
701         if (thispe .eq. 0) then
702             call fidabbdopt(lenrwbbd, leniwbbd, ngebbd)
703             write(*,28) tret, umax, iout(9), iout(3), iout(7),
704             & iout(20), iout(4), iout(16), ngebbd, rout(2),
705             & iout(18), iout(19)
706         28         format(' ', e10.4, ' ', e13.5, ' ', i1, ' ', i2,

```



```

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

```

```

766     parameter (nlocalg = mxsubg*mysubg)
767     double precision dx, dy, coeffx, coeffy, coeffxy
768     double precision uext((mxsubg+2)*(mysubg+2))
769 c
770 c local variables
771 c
772     double precision rtol, atol
773 c
774     common /pcom/ dx, dy, coeffx, coeffy, coeffxy, uext,
775 &                nlocal, neq, mx, my, mxsub, mysub, npex, npex,
776 &                ixsub, jysub, thispe
777 c
778     write(*,30) mx, my, neq, mxsub, mysub, npex, npex, rtol, atol
779 30    format(/'fidakryx_bbd_p: Heat equation, parallel example problem',
780 &          ' for FIDA', /, 16x,'Discretized heat equation',
781 &          ' on 2D unit square.', /, 16x,'Zero boundary',
782 &          ' conditions, polynomial conditions.', /,
783 &          16x,'Mesh dimensions: ', i2, ' x ', i2,
784 &          '          Total system size: ', i3, '//,
785 &          'Subgrid dimensions: ', i2, ' x ', i2,
786 &          '          Processor array: ', i2, ' x ', i2, /,
787 &          'Tolerance parameters: rtol = ', e8.2, '   atol = ',
788 &          e8.2, /, 'Constraints set to force all solution',
789 &          ' components >= 0.', /, 'SUPPRESSALG = TRUE to remove',
790 &          ' boundary components from the error test.', /,
791 &          'Linear solver: SPGMR.   Preconditioner: BBDPRE - ',
792 &          'Banded-block-diagonal.')
```

```

793 c
794     return
795 end
796 c
797 c =====
798 c
799     subroutine prntcase(num, mudq, mukeep)
800 c
801 c local variables
802 c
803     integer*4 mudq, mukeep
804     integer num
805 c
806     write(*,31) num, mudq, mukeep
807 31    format(/, 'Case ', i2, /, '   Difference quotient half-',
808 &          'bandwidths =', i2, /, '   Retained matrix half-bandwidths =',
809 &          i2, //, 'Output Summary',/, '   umax = max-norm of solution',
810 &          /, '   nre = nre + nreLS (total number of RES evals.)',
811 &          //, '   time          umax          k nst nni nli nre',
812 &          '   nge          h          npe nps', /,
813 &          '-----',
814 &          '-----')
```

```

815 c
816     return
817 end
818 c
819 c =====
820 c
821     subroutine prntfinalstats(iout)
822 c
823 c local variables
824 c

```

```

825         integer*4 iout(*)
826     c
827         write(*,32) iout(5), iout(6), iout(21)
828     32    format(/, 'Error test failures          =', i3, /,
829            &      'Nonlinear convergence failures =', i3, /,
830            &      'Linear convergence failures    =', i3)
831     c
832         return
833     end

```

